NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

CONSTRUCTING A REAL-TIME MOBILE ROBOT SOFTWARE SYSTEM

by

Kevin LaMonte Huggins

September 1994

Thesis Advisor: Yutaka Kanayama

Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	September 1994	Master's Thesis

4. TITLE AND SUBTITLE
Constructing a Real-Time Mobile Robot Software System (U)

5. FUNDING NUMBERS

6. AUTHOR(S)
Huggins, Kevin LaMonte

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the United States Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The problem with the Model-based Mobile robot Language(MML) processor is that the code is unstructured, causing the system to be unstable; it is very difficult to read because of deficient source code documentation; and because of poorly defined function interfaces and extensive functional coupling, the system is hard to maintain.

To fix the MML processor, we performed a manual static analysis of the existing source code to understand its structure. Next, based on the analysis, the software system was restructured and the functionality enhanced. Finally, explicit source code documentation was added in the form of comments.

There are several results with the new system. First, global variables are reduced from 152 to zero. Secondly, function interfaces are clearly defined and function coupling is enhanced. Finally, the source code is extensively documented. Following from these results, the new system is more stable, easier to read and understand, and simpler to modify.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Autonomous vehicle, robot, software engineering, real-time system	107
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

Approved for public release; distribution is unlimited

CONSTRUCTING A REAL-TIME MOBILE ROBOT SOFTWARE SYSTEM

by

Kevin LaMonte Huggins
Captain, United States Army
B.S., United States Military Academy, 1986

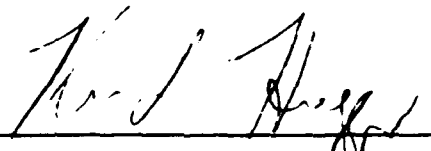
Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL

September 1994

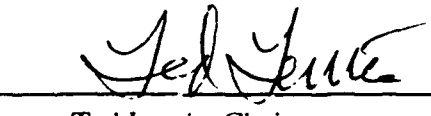
Author: 

Kevin LaMonte Huggins

Approved By: 

Yutaka Kanayama, Thesis Advisor

Frank Kelbe, Second Reader



Ted Lewis, Chairman,
Department of Computer Science

ABSTRACT

The problem with the Model-based Mobile robot Language(MML) processor is that the code is unstructured, causing the system to be unstable; it is very difficult to read because of deficient source code documentation; and because of poorly defined function interfaces and extensive functional coupling, the system is hard to maintain.

To fix the MML processor, we performed a manual static analysis of the existing source code to understand its structure. Based on the analysis, the software system was restructured and the functionality enhanced. Finally, explicit source code documentation was added in the form of comments.

There are several results with the new system. First, global variables are reduced from 152 to zero. Secondly, function interfaces are clearly defined and function coupling is enhanced. Finally, the source code is extensively documented. Following from these results, the new system is more stable, easier to read and understand, and simpler to modify.

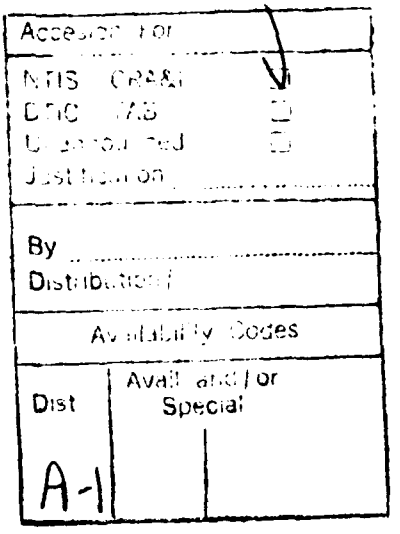


TABLE OF CONTENTS

LIST OF FIGURES

I. INTRODUCTION

With the changing world environment, the military's structure has changed to meet those needs. In particular, with the demise of the USSR as a threat, US military forces have decreased significantly in the recent past. With this down-sizing, the military is now required to do more with less. This requirement has led the armed forces to rely more on automation to fill the gap of reduced personnel and equipment. One key element to this move to more automation, is autonomous vehicles. These vehicles will continue to play a greater role in this nation's defense. At the Naval Postgraduate School (NPS), the Yamabico robot is an example of active research in the area of land based autonomous mobile robots.

Yamabico is a real-time mobile robot that is able to sense its surrounding and plan its motion. The software system that supports Yamabico is called the Model based Mobile robot Language (MML). As an on going research project, many people have contributed to research over the years. Unfortunately, software engineering practices have not always been followed. As a result, MML has become very unstable and difficult to maintain, causing a hinderance to research.

A. PROBLEM STATEMENT

The problem addressed in this thesis is how to construct a software system for an autonomous mobile robot that is stable, readable and modifiable. Specifically, this research is an improvement of the original Yamabico software system named MML10. In our work, the motion control rules are developed to provide path planning, as with various supporting modules such as the geometric functions and sequential commands. The software system developed as the result of this research is called MML11.

B. YAMABICO BACKGROUND

The Yamabico-11 mobile robot translates in two dimensional space and is controlled by MML. The Yamabico architecture consists of several systems. They are the CPU,

wheels, sonar, and Input/Output systems. In the next few paragraphs we will give a brief overview of these systems, however, for more in-depth information, see (Scott, 93) and (Book, 94).

The CPU system consists of a SPARC-4 based mother-board, with power supplied by two 12 volt motorcycle batteries. One communicates with the robot either through a 9600 baud port connected to a laptop MacIntosh Power Book, or a telnet connection from a Sun work station. Compilation and downloading can be accomplished from any Sun work station. Once the kernel is made, there is a process that automatically copies the kernel to a holding directory. When the command is given to download the kernel, either from a Sun work station or the laptop computer on the robot, the kernel is then downloaded over an ethernet connection.

The wheels system includes a VME bus interface card that provides the user a means of communication with the wheels system. Power is provided by two DC motors and a motor control circuit card manipulates the motors based on information from the interface card. Finally, the shaft encoders provide information that helps determine the speed and distance travelled. (Scott, 93)

The sonar system is used to gather sonar information from the forward, rear, lateral, and diagonal directions. This system consists of twelve 40kHz ultrasonic sensors. The sonar interface card is used to collect information and control the sensors. (Book, 94)

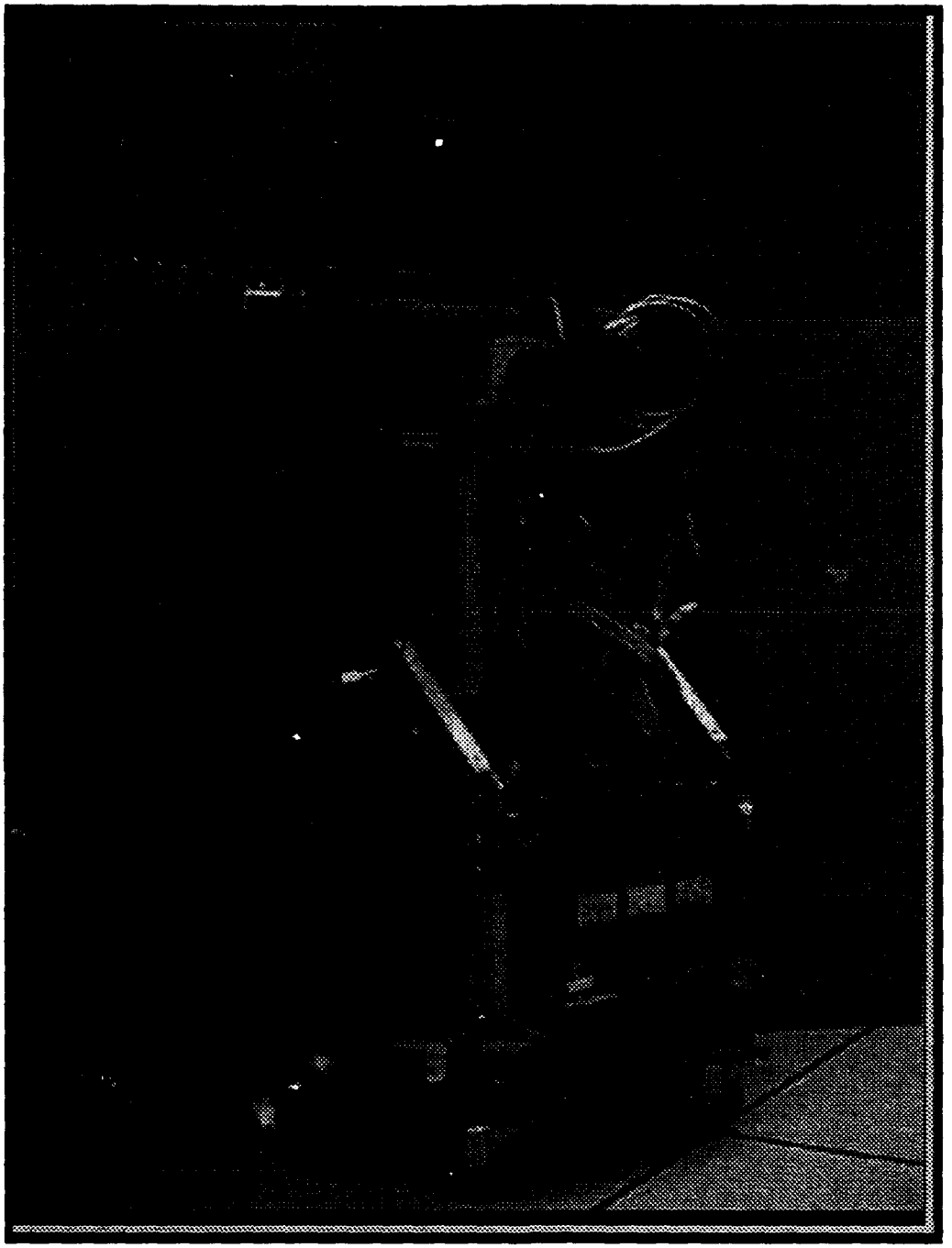


Figure 1: Yamabico-11 Mobile Robot

C. MML BACKGROUND

MML is a high level software system that controls the Yamabico robot. It's design goal is to be a general purpose standard language for autonomous mobile robots, independent of any physical attributes of the robot.

The hardware interrupts are used to provide the real-time aspect of MML. With these interrupts, background processes achieve pseudo-concurrent processing. It is pseudo-concurrent and not fully concurrent because no code for a process is ever interleaved with another, because processes are not allowed to share the same priority. Hence, processes are interleaved, but their code is not. Therefore, this type of process concurrence is limited to the interrupts provided by the CPU architecture. (Scott, 93)

The MML architecture is rather simple conceptually. It is has two process levels: foreground and background. The user program runs in the foreground and the motion control functions operate in the background. There are two sets of function libraries that provide the user with control of the robot. They are called immediate and sequential

functions. Immediate functions are executed immediately when they are read. However, sequential functions are first stored in an instruction buffer and executed sequentially.

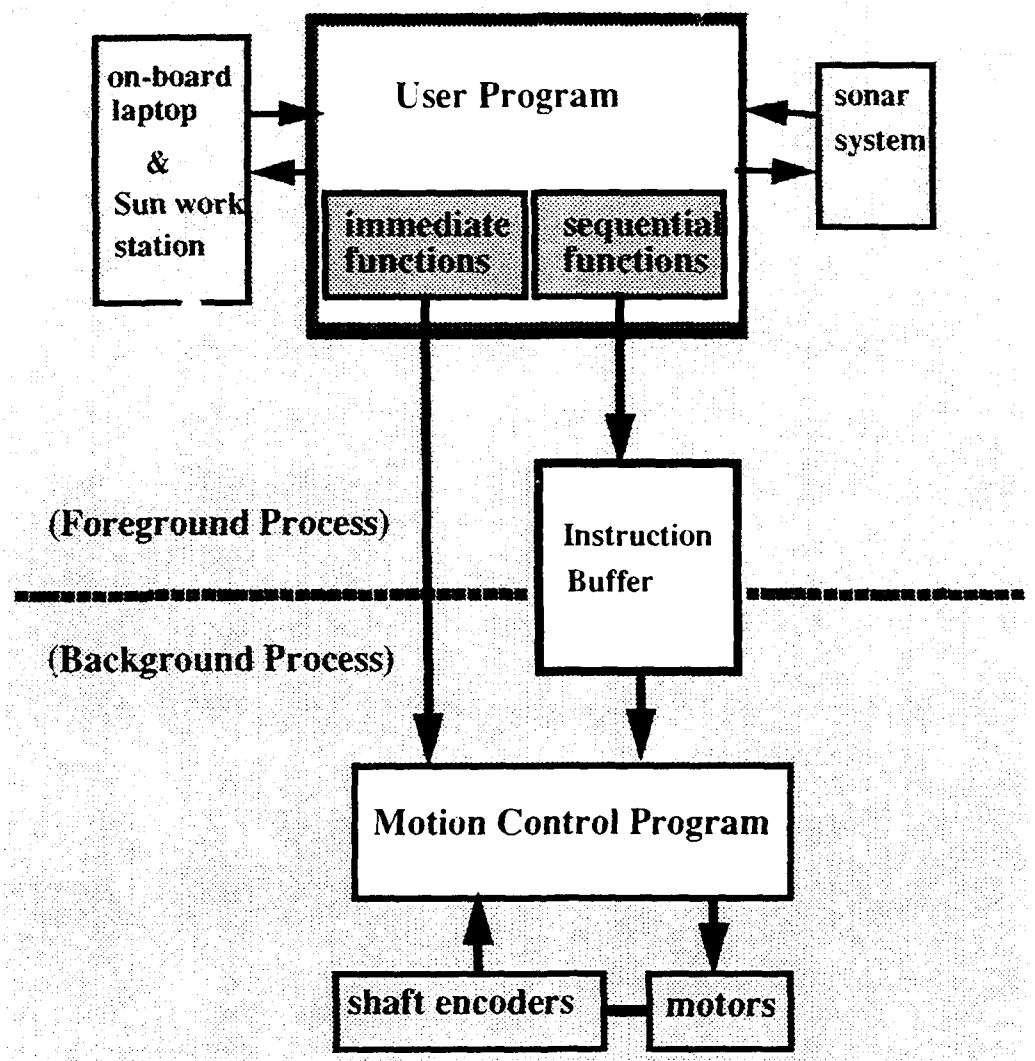


Figure 2: MML System after (Lee, 93)

Recent developments have significantly improved various aspects of the MML system. First, Scott Book, in his research, developed a solid core system that localized hardware dependencies, and reduced assembly code to a single module (Book, 1994). Frank Kelbe, the Yamabico group software design coodinator, designed and implemented data tracing functions for both the motion control and sonar modules. He also redesigned the instruction buffer module and implemented timer functions.

D. ORGANIZATION OF THESIS

This thesis is organized around the construction steps taken for the new version of MML. First, in the next chapter, we describe software attributes that are desirable in any software system. Then in Chapter III, we analyze the original Yamabico code to understand its structure. From this analysis, we develope a methodology in Chapter IV to constructing MML11. Next, in Chapter V, we document the implementation phase, describing major milestones reached in the process. We describe the resultant function specifications in Chapter VI, and discuss the results in Chapter VII. Finally, we conclude in Chapter VIII with a summary and recommendations for future research.

II. DESIRABLE SOFTWARE ATTRIBUTES

For any software system, there are certain attributes that are desirable. They provide a common standard by which all software systems can be compared to and judged. These attributes are cohesion, coupling, modifiability, modularity, readability, and robustness. An important term to understand when applying these attributes is the module. One must consider logical relationships and physical storage properties in order to describe a module. First, a module is a logical break down of routines that have a common functionality or who's functionalities are related. In reference to physical storage, modules are best if kept as small as possible; one module per file(Scott, 93).

Cohesion measures the strength of relationships between code segments in the same module. Two code segments are related if they reference the same element. Accordingly, it is desirable for related segments to be collected in the same module to the element referenced. Coupling describes the strength of references between modules. As coupling increases, maintainability and readability decreases. There are two types of coupling: common and control. Common coupling applies to references to internal data elements or data structures by another module. Control coupling occurs when flags, used to modify the behavior or actions of a routine, are used between modules. (Scott, 93) and (Book, 94)

A system is modifiable if changes can be made to one segment of code without generating adverse side effects in another segment. The degree that a system is modifiable is directly related to the system's measure of coupling and cohesiveness. Once produced, a modifiable system is easily changed and maintained. Modularity is defined as the partitioning of the system into small segments. There are two major goals associated with modularity. First, this process is to design each segment around a particular logical function performed by the system. The result is a system that has strong cohesion. The second goal is to minimize the amount of coupling. This is done by using a clean and concise interface. (Book, 94)

In the Yamabico research group, as with others, often the person who implemented part of the system is no longer available. For this reason, it is crucial that systems are readable. A system is readable when it exhibits the same behavior during operation as expressed in the source code. Small modules with independent, well defined and clearly documented behavior are the most readable (Scott, 93). Systems that can detect errors and recover are considered robust. This requires the addition of exception handling functions and procedures to the system. These code segments allow the system to process exceptional conditions such as division by zero or value out of limits (Scott, 93).

III. ANALYSIS OF THE EXISTING SYSTEM

In this chapter we present two techniques used to help describe the structure of MML10. This description of the original system helps in the design of MML11. Specifically, we take the results of the analysis in this chapter, and use it as a starting point for our design goals. The desirable attributes discussed in the previous chapter are the ending point. We then report the results of the analysis on the code. This information will be used in the next chapter to develop a design philosophy.

A. CALL-HIERARCHY TRACING

Call-hierarchy tracing is an analysis technique to describe function categories. It is used for identifying whether a called function's scope is local or global. This tracing method also provides a physical picture of the present software system. Finally, it helps in determining where to place functions. Function calls fall into one of three categories. They are either local, module, or global.

A local category happens when a function is called by only one other function. In Figure 3, which is an example of the local category, function B is called only by function A. With this category, one can place both functions in the same module. If a function is

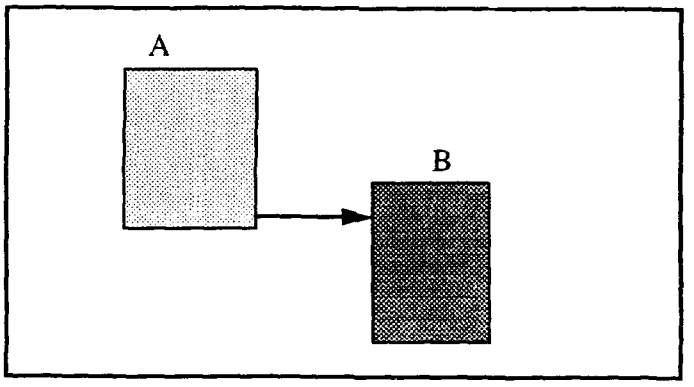


Figure 3: Local category

called by several other functions that all have related operations, they can be placed in the same module. This type of function call falls into the module category, an example of which is given in Figure 4.

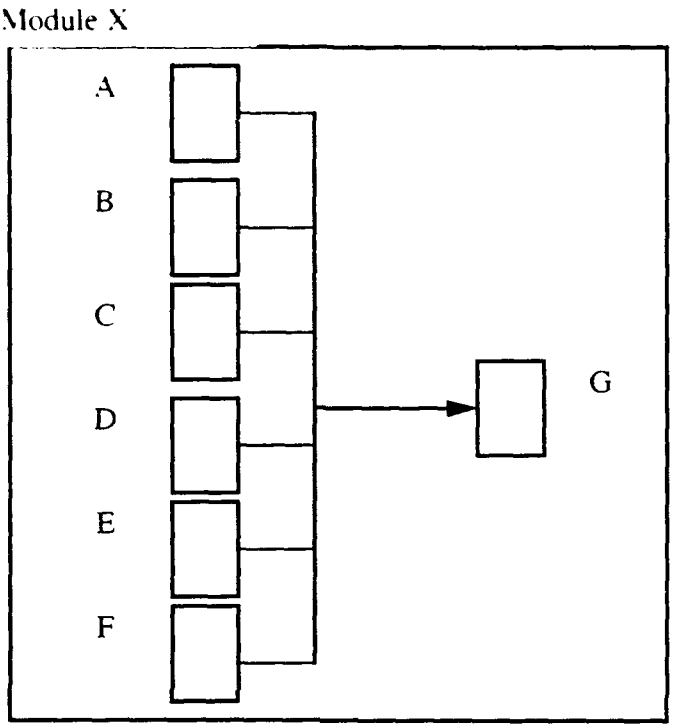


Figure 4: Module category

Finally, the global category occurs when a function is called by several functions that have unrelated operations. An example of a global category function is given in Figure3, with functions A-C, which are in different modules, call function D. Global category

functions are usually found in libraries of utility functions. (Scott, 93) An example in

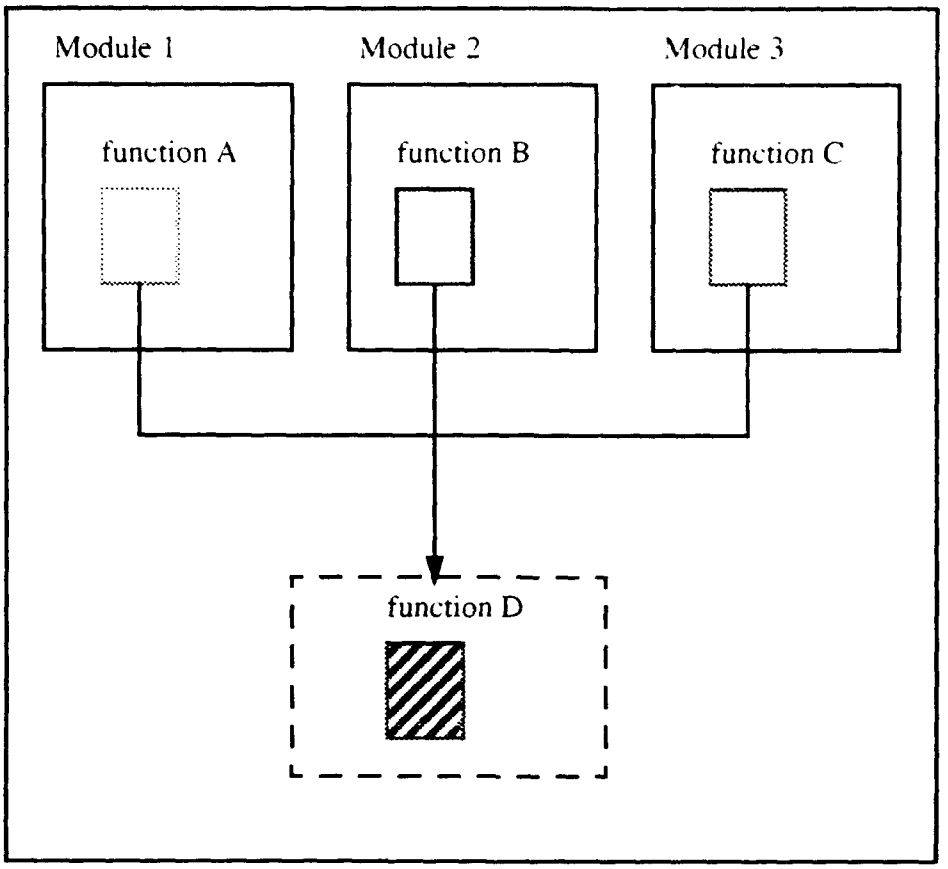


Figure 5: Global category

MML is the geometry file which has all of the geometric utility functions used by several different modules.

B. GLOBAL VARIABLE TRACING

Global variable tracing is used to categorize global variables in a software system. Variables are either module or globally visible. These types of variables are not passed into a function as a parameter nor declared locally. Globals need to be analyzed to see whether they should stay global, local to a module, converted to a locally declared function variable,

or passed in as a parameter. The goal is to minimize the use of global variables. The benefit of reduced globals is increased readability and reliability, because of better defined interfaces (Scott, 93).

C. RESULTS OF ANALYSIS

There are several interesting results from the call-hierarchy and global variable tracing analysis of MML10. First, there is an extensive use of globals variables used throughout the entire system. Specifically, there are 152 global variables, with all motion control variables being global. Secondly, all functions are globally visible. There are no static definitions of variables or functions. As a result, all variables and functions defined have a global scope: they can be seen and accessed by any other function in the system.

One significant implication is that function interfaces are blurred. Instead of passing or locally declaring variables, most functions simply access and modify variables directly. This directly influences coupling. Since code segments from different modules commonly reference the same element, coupling is tight. Figure 6 provides an example of this challenge in MML10. The function that calculates the commanded velocity, get_veloctiy(), in the motion control module, references elements in the instruction buffer

module directly without using a function interface. Accordingly, this causes a strong

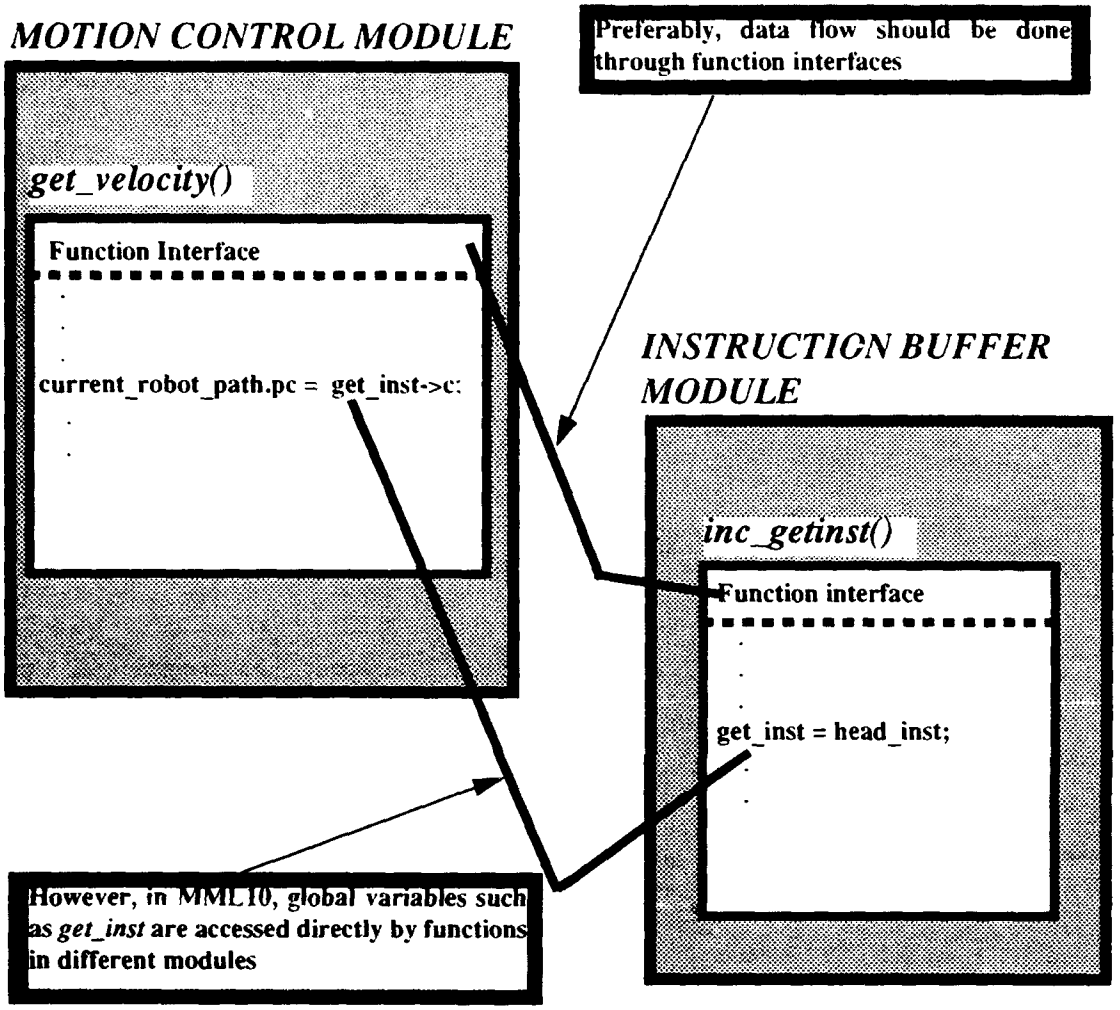


Figure 6: Example of blurred function interfaces in MML10

dependency between these two distinct modules. Since function interfaces are blurred by the extensive use of global variables, function cohesion suffers also. For example, code that enables the robot to rotate is found in the motion control module and the instruction buffer module. However, this functionality belongs in one module.

Because of its structure, modifiability is very difficult. One has to understand the entire system prior to making any changes because an element manipulated in one module

may affect or be affected by another potentially unrelated function in another module. Since all functions and variables are visible globally, one has to check the entire system prior to making modifications. This is not only tedious and time consuming, it makes modifying the code a much more difficult task that it should be.

IV. METHODOLOGY

From the last chapter we understand the present structure of MML10 and its shortcomings. We now take that information while considering the desired software attributes, and in this chapter develop a methodology for constructing MML11. Although all software attributes are important in a software system, some are more pertinent depending on the particular situation. With MML, the environment that it is used in significantly influences which software attributes need to be the center of focus.

The Yamabico research group is a dynamic environment. Members are continuously joining or moving on to other projects. As a result, it is important to design a system with the following assumptions: 1)Different people will be working on various parts of the system at different times and 2) Few people have a need to know and understand the entire system.

Understanding these assumptions, we can focus our attention on those software attributes that best support our design assumptions. Specifically, we are most interested in code that has excellent readability, modifiability, and coupling attributes. It is essential that the code is very readable. Since different people are working on the code at different times, one can not assume that the author of a particular code segment will be available to explain its functions. Therefore, documentation is the critical link. Since one of the first steps in modifying code is understanding what it does presently, documentation also enhances modifiability. Finally, by having a software system that is easily modifiable, the research process is enhanced.

With people specializing in different subsystems of MML, it is important that the variables and code modified in their areas do not have unexpected negative effects on routines in other unrelated parts of the system. A software system that displays loose coupling provides this type of environment.

To obtain a system with the above mentioned desirable software attributes, design goals are needed as a guide in the development of the new system. Our design goals are

three-fold. They are the elimination of global variables, clearly defined function interfaces, and highly explanatory documentation. These goals will best guide the development of MML11 to obtain the software attributes and support our design assumptions.

A. GLOBAL VARIABLES

Our biggest and most challenging task is the removal of global variables from MML10. In this version of MML, the structure of the software system is driven by procedural control. In other words, the key decision making factor is how the flow of control is effected. This emphasis on flow of control minimizes the focus on data grouping. If your code is developed with the grouping of data as the priority, you place your functions in modules where they can best access the needed data for their operations. However, the easiest way to develop a system where flow of control is the priority, is to have the least amount of restriction on data accessibility. This is done in MML10 by making variables global. Additionally, with these variables being global, they are initialized in one location, the system main module.

Understanding the present challenges with global variables in MML10 presented in the previous chapter, our approach is focused more on the grouping of data. Since we want to eliminate global variables, all variables have to be either declared locally in a function, or module. To accomplish this, we have to first determine how we to group the data. Initially, we consider breaking the data down into hardware and system related variables. However, this quickly proves to be unmanageable because there are so many variables that do not fit into either of these categories very well.

We next attempt to group variables based on the frequency they are used in a file. We would declare a variable in a module that has the most references to that variable. A tool that is of great assistance is the unix grep command. This allows us to count the times that the variable appears in a file. There are three cases that we face when attempting to group these variables. In all three cases, the variables are initialized in the main module of MML10. First, in the easy case, when a variable is used only in one file, we statically

declare the variable locally to that file. Next, the more difficult case is when a variable is used mostly in one file but also found in others. In this case, we still statically declare the variable in the file where it is used the most. We then write function interfaces to allow functions from the other modules to access the variable. The most difficult case, however, is when different modules reference the same variable with about the same frequency. In this case, we go into each module and determine how critical the variable is to calculations in the local functions. For example, there could be two modules that reference a variable the same amount of times. However, module 1 modifies the variable while module 2 references the variable. In such cases, the variable is more critical in module 1 because it is not only accessed, but updated. Accordingly, the variable is placed in module 1 with function interfaces written in module 1 for access by module 2.

After all variables are grouped into modules, the process is repeated at the module level to determine whether a variable would stay visible at the module level or be declared in a function. If the variable is only used in one function, then it is declared there. However, when a variable is used by several functions, it retains its module level visibility. For example, in the motion control module, this method is used to determine whether the variable that held the value for the current vehicle configuration, (named *vehicle*) would remain at module level or be declared in a function. After analyzing the code, we find that several functions in the motion control module use *vehicle*. For this reason, *vehicle* is declared local to the entire motion control module.

With all global variables removed, we still need to initialize variables. Different from MML10, we require an initialization routine in each module in MML11. Although this requires more code than the approach used in MML10, initializations are now easier to manage. We are not concerned with a huge file of variable initializations anymore, but instead with several smaller routines that are specifically related to the file that they support. This makes maintaining and modifying them much easier.

B. FUNCTION INTERFACES

In earlier versions of MML, data access can not be controlled through function interfaces because global variables usurp the need to pass parameters. As a result, critical regions are vulnerable. Specifically, a function can be interrupted while updating a variable because critical regions are not controlled. However, the goal with MML11 is to have all data in functions to be either locally declared, passed in as a parameter, or local to the module. Inter-module communications therefore, is limited to function interfaces. This additional control in MML11 ensures control in critical regions because variables can only be accessed through a function interface. With this limited access, interrupts can be disabled if deemed necessary, within the function.

This design goal has a significant impact on the structure of MML11. In MML10, immediate functions are grouped together into one file. Since few parameters are passed in the older version of MML and data grouping is not a concern, it is logical that these functions are placed in the same file. However, in the design of MML11, we find that this structure is unacceptable. The immediate functions fall basically into two categories. They either work with data that is updated by the motion control module, or they maintain data that is only assessed by motion control and other modules. As a result, we group the functions that access variables that are updated by the motion control module, into motion control. In this way, any other module that needs these variables access them through function interfaces in motion control. The alternative would have been to maintain the original structure with all of the immediate functions in one file. Function interfaces would still have been written in the motion control module to provide access to variables for those immediate functions that need them. We then would have to call these functions from within the immediate function. The method we use thus allows us to cut out the extra unneeded step in accessing variables that are controlled by motion control. The immediate functions that do not access variables controlled by motion control are still kept in a separate file.

C. DOCUMENTATION

Our review of MML10 greatly influences our documentation goals for MML11. We found it very challenging to understand MML10 without someone available who is already familiar with the code. Functions usually have sparse comments if any. Variable naming also makes it difficult to follow because they are usually very short and non-descriptive. Therefore, with MML11, following our assumption that people who wrote the code will not be available, we decide to thoroughly comment our code and provide descriptive file, function and variable names.

First, a naming convention has to be chosen. We decide to adopt one that has recently been designed for the Yamabico research group, because of its simplicity and ease in duplication. Overall, more descriptive names are used. Full words are used as much as possible, and names are chosen that best describe what exactly the variable, function or constants does. For variables, the first letter is lower case. If more than one word is used in the variable name, the first letter of each word other than the first word is capitalized. The capital letters are used to distinguish the words instead of underscores. The benefit is that it requires less space for variable names. Function names are the same as variables.

Constants are written as all capitals. To distinguish between words if more than one word is used for the constant name, underscores are used.

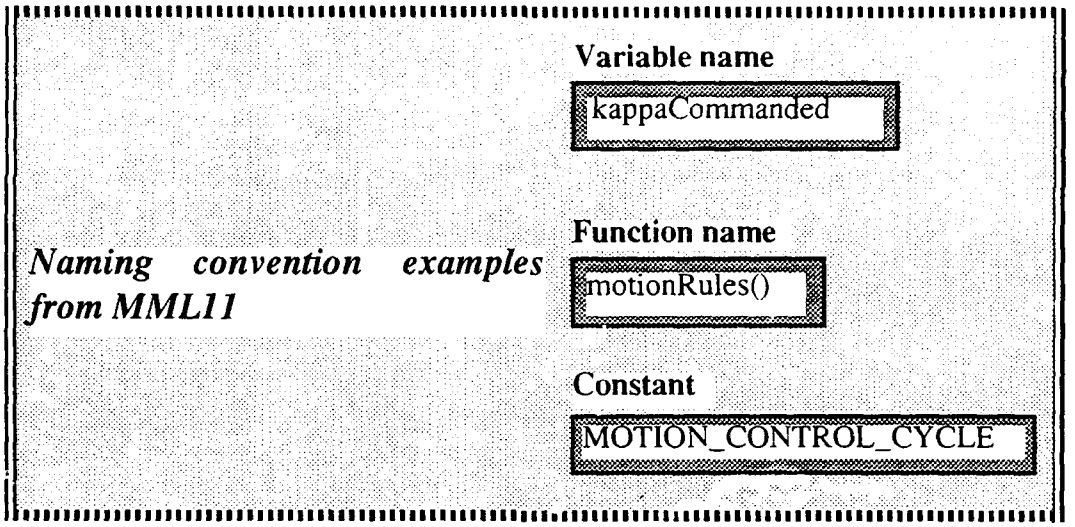


Figure 7: Naming convention

Our comments are written with the assumption that the reader is unfamiliar with MML. Also, we provide descriptive names to variables and functions so that their names describe what they do. Each declared variable in a module is documented. Functions also are extensively documented, not only to explain the purpose, but, if appropriate, to give normal uses of the functions. In critical areas, function calls were documented to help the reader. For example, in the main routine of the motion control module, all of the steps of the motion control theory are represented. During each motion control interrupt, this routine is what is called. Because of its critical role in the MML system, each line of this routine is documented to help guide the reader through the module. For example, for each function call, we identify in which module the called function is located.

All locally declared static function prototypes are provided brief documentation so that a reader does not have to search through the entire module to find the full documentation. Function prototypes also provide an added benefit of argument checking. MML10 is written in a non-standard version of the C language. Function prototypes, in this version, do not require an argument to listed in the parameter list, even if arguments are

passed into the function. The advantage of using prototypes under ANSI C is that they allow the complier to perform compile-time type checking on the arguments of a function. The compiler will check all calls to the function to make sure it has both the correct number and types. Depending on the severity, the compiler may issue a simple warning, such as when an integer is expected, but a chararacter is passed. An error may be issued, instead, for mistakes such as the wrong number of parameters. This type checking simply makes the code more reliable by having the compiler find problems before testing. (Kelbe, 94)

```
/* calculates the vehicle's next configuration based on the distance
      travelled in the last motion control cycle */
static CONFIGURATION
localize(CONFIGURATION robot, double deltaS, double deltaTheta);

/* calculates the next commanded linear velocity value. */
static double getLinearVelocity(double actualVelocity,
                                double lastCommandedVelocity);

/* calculates the distance remaining on a path to reach a configuration.
   Used with bline calculation. */
static double restOfPath(void);

/* determines whether the vehicle needs to decelerate.
   Used in bline calculation */
static int needsToDecelerate(double actualVelocity);
```

Figure 8: Static function prototype documentation examples

Function prototypes in our header files are documented similar to functions in our program files. Specifically, the commented section above the function name in the program files is used in the header file prototype. We do this to provide the most complete documentation for the reader in the header file. Our goal is for a reader to be able to understand and apply functions found in a file simply from reading the header file.

V. IMPLEMENTATION

A. DATA STRUCTURES

With the re-engineering of MML, fewer software structures were needed. Most of MML was described using only five basic data structures. In this section we explain these structures.

1. Point

The POINT structure is used to describe a position in two dimensional cartesian coordinates. The structure is made up of two doubles named X andY.

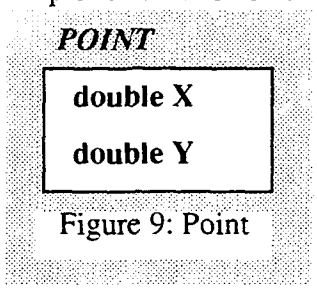


Figure 9: Point

2. Velocity

The VELOCITY structure is used to describe a two dimensional velocity vector. The data structure is made up of two doubles that represent the linear and rotational elements of velocity. They are appropriately named Linear and Rotational, respectively, in the VELOCITY structure.

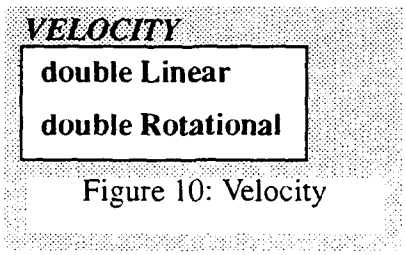


Figure 10: Velocity

3. Configuration

The CONFIGURATION is the standard structure for describing location and direction for an object. It consist of Posit, which is of type POINT, which identifies an objects position in two dimensional cartesian coordinates. Next, there is a double called

Theta that describe's the object's orientation in relation to the X coordinate. Finally, there is another double in the CONFIGURATION structure called Kappa that represents the curvature of an object's path.

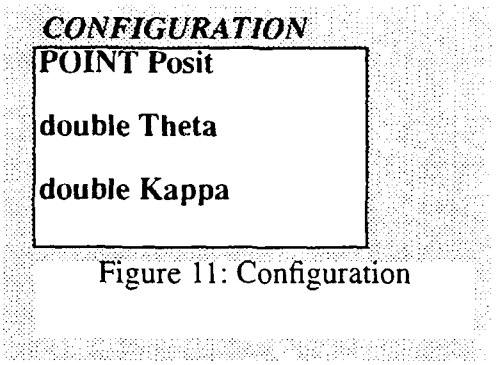


Figure 11: Configuration

4. Parabola

The PARA structure is used to describe a parabola. A common use of the PARA data structure is as a configuration that the robot follows. Specifically, it is another type of path that is available in the library of functions for motion types. It consist of a Focus which is of type POINT and a Directrix which is of type CONFIGURATION.

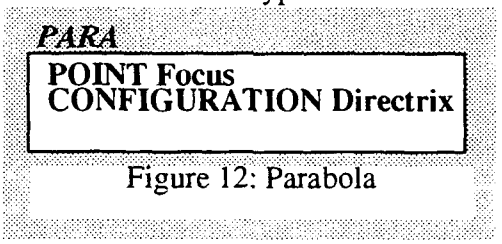


Figure 12: Parabola

5. Path Element

The PATH_ELEMENT data structure is used to describe and store the various types of movements. This data structure consist of config which is of type CONFIGURATION. It holds the configuration of the path that the robot is to follow. PATH_ELEMENT also contains pathType, which is of type PATH_TYPE. A PATH_TYPE is a data structure used to identify the various paths that are available to the robot. It consist of the mode which is of type MODE and class which is of type CLASS. Type MODE is an enumeration type that gives a name to each path that the robot follows. Presently, the modes that are available include NOMODE, STOPMODE, PATHMODE,

ROTATEMODE, KSPIRALMODE, and PARAMODE. Type CLASS, which is also an enumeration type, is used to name and categorize the various PATHMODE types. The list of classes include NOCLASS, LINECLASS, CIRCLECLASS, and BLINECLASS.

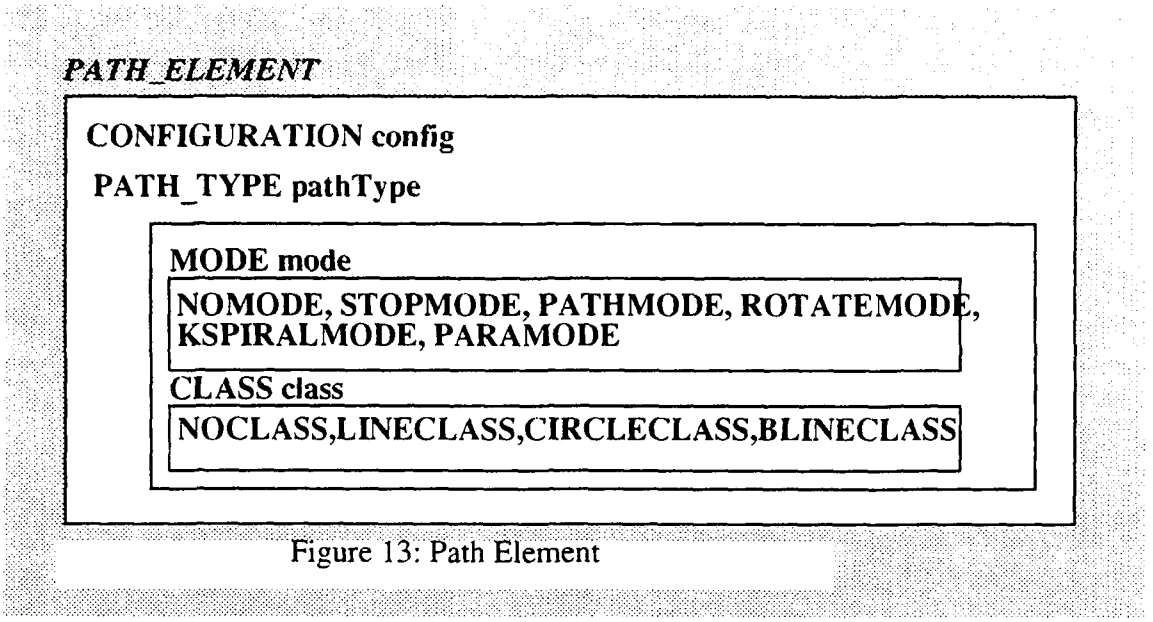


Figure 13: Path Element

6. Significance

With the above mentioned data structures, all motions that are presently performed by the Yamabico robot, can be described and represented. More importantly, because of the modular design, additions and maintenance of these data structures is simplified

B. GEOMETRIC FUNCTIONS

1. Pointers Use Reduced

Pointers are used extensively in the original version of MML. This is because the language that it was written in, a non-ANSI C, does not support structure passing. As a result, one would need the following type of function interfaces:

CONFIGURATION * defineConfig(POINT position, double theta, double kappa, CONFIGURATION *configPtr)

Notice that a pointer to a CONFIGURATION was returned. However, a pointer to a CONFIGURATION was also passed in. At first glance, we assume that the pointer to the CONFIGURATION that is passed in, is a duplicated effort because one should just be able to declare a a CONFIGURATION type in the function defineConfig and return its address. However, this would cause a logical error because once the function returns, the value that the pointer is pointing to is discarded with the function. One is left pointing to garbage. So without the ability to pass structures, it is very awkward handling them in function calls. However, with the ability to pass the structures, such as a CONFIGURATION type, the above function interface is written as follows:

CONFIGURATION defineConfig(POINT position, double theta, double kappa)

As one can see, the interface is simpler, and easier to read and understand.

2. Duplicated Functions

In the original version of MML, there are functions that are duplicated. Although they have different names, they perform exactly the same function. In the new version of MML, the duplicated functions are eliminated. One example are the normalize functions. There were two called normalize() and norm(). After inspection, we determined that they perform the same task and have the same logic.

3. Functions Added

There are also geometric functions that are new to the updated version of MML. They are included to support new functionalities added to MML. For example, there is a function that calculates the circular arc. This function is used in conjunction with the composition function to create a new way to localize the position of the robot during its motion control cycle interrupt. This method is cleaner and more efficient. Also, a function to determine the signed difference between a point and a configuration, is added. This is included mainly to support the parabola calculations, however is available to all modules.

C. MOTION CONTROL

Among other things, the design goals would help produce a software system that is easily maintainable. This is very crucial in an ongoing research project such as Yamabico. One area that is critical is the means of representing and categorizing motion control commands. The use of modes and classes is used as a means of describing the various control commands. The mode describes the basic types of motions commands. They include path, stop, rotate, kspiral, and parabola modes presently. Path modes are further broken down into classes. These class descriptions presently include line, circle and backward line path tracking. The benefit of using this taxonomy is that it encourages a modular design. Specifically, when new motion commands need to be added to the library of commands, a new mode or class only needs to be added.

1. Line

In our implementation of these commands, the simplest case--the line-- was chosen first. Sequential commands provide a means of putting motion commands into the instruction buffer, to be later extracted sequentially and passed to the motion control module. However, since the sequential commands had not been implemented yet, it was necessary to hard code the line command directly into the motion control module. With this done, we first tested the robot by allowing it to track a line that was configured parallel to the robot's configuration with no offset. This enabled us to ensure that the robot could follow a straight line. When this was accomplished, we allowed the robot to follow a line that was configured parallel to the robot but offset by thirty centimeters. With this test, we confirmed that the robot could correct it's path to follow a line that was offset. Finally, we commanded the robot to follow a line that was not only offset but not parallel. When this was accomplished, testing for the simple line case was complete.

2. Bline

The backward line (bline) tracking command gives the robot not only a configuration to follow, but a location to stop. With the bline command, the robot will stop

at the location described by the bline configuration. During our implementation, we initially had challenges with the robot not stopping. This was because we had overlooked changing the mode to stop mode (thus commanding the robot to stop) when the robot had reached the bline configuration. Once this correction was implemented, the bline command worked as expected.

D. SEQUENTIAL COMMANDS

As discussed earlier, sequential commands were a library of functions that allow the user to interface with the robot. When commands were issued, they were placed into the instruction buffer. They were then extracted and executed sequentially. Our plan was to test the line and bline commands called sequentially using the sequential commands and the instruction buffer. First, we tested the sequential line command. Our only challenges centered around properly initializing functions. This challenge continued throughout our entire implementation phase. With the line case, the instruction buffer module was not being initialized. Next the bline sequential command was implemented with no significant challenges. The testing of theses commands was broken down into three stages. First, the robot transitioned from one line that it was following to another. Second, the robot had to transition from a line to a bline. Finally, it was tested from a bline to another bline.

VI. FUNCTION SPECIFICATIONS

In this chapter we list the user and system functions of MML11. With each specification, we provide the ANSI C sysntax for the function and a description of the function.

A. USER FUNCTIONS

1. Set Robot's Configuration (Immediately)

Syntax: void setRobotConfigImm(CONFIGURATION)

Description:

This function immediately sets/updates the robot's configuration. This can be done while the robot is at rest or moving.

2. Get Robot's Configuration

Syntax: CONFIGURATION getRobotConfig(void)

Description:

This function returns the current configuration of the robot.

3. Set Configuration (Immediately)

Syntax: void setConfigImm(CONFIGURATION NewConfig)

Description:

This function enables the user to update the robot's position and theta. However, the Kappa is not adjusted with this command.

4. Track a Line

Syntax: void line(CONFIGURATION config)

Description:

Basically the robot follows a directed path element defined by the configuration that is passed as a parameter. The robot leaves this element when it comes to a transition point or when an immediate motion function are called. The robot's speed is automatically

reduced to allow the robot to make sharp turns. This is reflected by the dependency between Kappa and the robot's speed. In simple terms, the robot's speed must be reduced to allow it to move safely with larger values of Kappa.

5. Track a Backward Line Segment

Syntax: void bline(CONFIGURATION config)

Description:

The robot follows this directed path element.defined by the configuration that is passed as a parameter. The robot will track the line config until it passes config itself and will transfer to the next path segment. If there is no next path segment, the robot will start to slow down at the configuration config and eventually stop with the current acceleration rate. Precisely speaking, the robot leaves the segment config when the robot's image reaches config (or is downstream of config).

6. Stop the Robot

Syntax: void stop(void)

Description:

When this function is processed from the instruction buffer, it calls the immediate command of stop. In doing so, the rotational and linear goal velocities are set to zero.

7. Set the Robot's Configuration

Syntax: void setRobotConfig(CONFIGURATION config)

Description:

The robot's configuration is set to the value of the parameter config. This function is processed only if the robot is in a stopped position.

8. Stop the Robot(Immediately)

Syntax: void stopImm(void)

Description:

This function immediately updates the goal velocity to zero in order to stop the robot. The sequential command stop() calls stopImm() once the sequential command function pair is read from the instruction buffer. Also, users can call stopImm() directly.

9. Set Linear Velocity(Immediately)

Syntax: void setGoalLinVelImm(double linearVelocity)

Description:

This function sets the goal velocity that the robot will attempt to achieve when it is following a path. This sets the speed of the robot immediately.

10. Get Linear Velocity

Syntax: double getGoalLinVel(void)

Description:

This functio.. retrieves the current goal linear velocity that the robot is following.

11. Set Rotational Velocity(Immediately)

Syntax: void setGoalRotVelImm(double rotationalVeloctiy)

Description:

This function sets and updates the goal rotational velocity that the robot will attempt to achieve when it is following a path.

12. Get Rotational Velocity

Syntax: double getGoalRotVel(void)

Description:

This function retrieves the current goal rotational velocity that the robot is following.

13. Set Linear Acceleration(Immediately)

Syntax: void setGoalLinAccImm(double linearAcceleration)

Description:

This function sets and updates the goal linear acceleration that the robot will attempt to achieve when it is following a path.

14. Get Linear Acceleration

Syntax: double getGoalLinVel(void)

Description:

This function retrieves the current goal linear acceleration that the robot is following.

15. Set Rotational Acceleration(Immediately)

Syntax: void setGoalRotAccImm(double rotationalAcceleration)

Description:

This function sets and updates the goal rotational acceleration that the robot will attempt to achieve when it is following a path.

16. Get Rotational Acceleration

Syntax: double getGoalRotAcc(void)

Description:

This function retrieves the current goal rotational acceleration that the robot is following.

17. *Set Size Constant*(Immediately)

Syntax: void setSizeConstantImm(double SizeConstant)

Description:

This function sets the size constant which is used, among other things, to influence the sensitivity of the steering function.

18. Get Size Constant

Syntax: double getSizeConstant(void)

Description:

This function retrieves the current size constant that is being used in motion control.

19. Set Total Distance(Immediately)

Syntax: void setTotalDistanceImm(double distance)

Description:

This function sets the total distance travelled by the robot to the value passed as a parameter.

20. Get Total Distance

Syntax: double getTotalDistance(void)

Description:

This function returns the total distance travelled by the robot.

21. Halt Motion(Immediately)

Syntax: void haltMotionImm(void)

Description:

This function brings the robot to a rest. It is different from the stop functions in that it's purpose is a temporary halt with the assumption that you will continue or resume the motion. Accordingly, the original goal velocity is saved to be later used by the resume motion command to allow the robot to continue travelling at the same speed as it was travelling before it halted.

22. Resume Motion(Immediately)

Syntax: void resumeMotionImm(void)

Description:

This function is to be called only after a halt velocity command. It allows the robot to resume the speed it was travelling before the haltMotionImm() was given.

23. Parabola(Immediately)

Syntax: void parabolaImm(PARA newParabola)

Description:

This is the immediate function that commands the robot to follow the parabola configuration passed in the parameter.

24. Euclidean Distance

Syntax: double euDis(double x1, double y1, double x2 double y2))

Description:

This function computes the Euclidean distance between two given points

25. Normalize

Syntax: double norm(double angle)

Description:

This function, when given an angle in radian, returns a normalized angle between $-\pi$ and π. This is the most common normalizing function used in the system.

26. Positive Normalize

Syntax: double positiveNorm(double angle)

Description:

This function, when given an angle in radian, returns a normalized angle between 0 and 2π.

27. Negative Normalize

Syntax: double negativeNorm(double angle)

Description:

This function, when given an angle in radian, returns a normalized angle between -2π and 0.

28. Normalize PI/2

Syntax: double normPIover2(double angle)

Description:

This function normalizes the input angle between -PI/2 and PI/2. This was specifically designed for parabola tracking calculations.

29. Signed Difference

Syntax: double signedDiff(CONFIGURATION config, POINT pt)

Description:

The signed difference function calculates the size distance between a point and a configuration. If the value returned by the function is positive, that means that the point is to the left of the configuration. If the value returned is negative, then the point is to the right of the configuration.

30. Define Configuration

Syntax: CONFIGURATION defineConfig(double x, double y, double theta, double kappa)

Description:

When passed the values that define a configuration (x,y,theta, and kappa), this function allocates and assigns a configuration. It returns a configuration.

31. Define parabola

Syntax: PARA defineParabola(double xf,double yf, double xd, double yd, double td)

Description:

When passed the values that define a parabola (xf, yf, (the focal point), xd, yd, td (the directrix), this function allocates and assigns a parabola. It returns a pointer to a parabola.

32. Reverse orientation

Syntax: CONFIGURATION reverseOrientation(CONFIGURATION original)

Description:

The purpose of this function is to reverse the orientation of a given configuration by 180 degrees. You pass in the original configuration and then the reversed configuration is returned.

33. Find symmetric configuration

Syntax: CONFIGURATION findSymConfig(double a, double alpha)

Description:

This function finds the symmetric configuration of an original configuration. The parameter -- a -- is the distance from either configuration to the intersection of both lines determined by the two configurations. The parameter -- alpha -- is the angular difference between both orientations. One drawback to this function is that it is not possible to represent a symmetric configuration whose alpha is equal to. FindSymConfig1() is used to cover these situations.

34. Find symmetric configuration 1

Syntax: CONFIGURATION findSymConfig1(double a, double alpha)

Description:

This function performs the same operation as findSymConfig(), except that it overcomes the drawback of not being able to handle the situation when alpha equals.

35. Inverse

Syntax: CONFIGURATION inverse(CONFIGURATION original)

Description:

The purpose of this function is to calculate the inverse of a given configuration such that: original * inversed = Identity. The parameter --original -- is the original configuration in global coordinates. This function returns the inversed configuration.

36. Compose

Syntax: CONFIGURATION compose(CONFIGURATION *first, CONFIGURATION *second)

Description:

The purpose of this function is to calculate the composition of two configurations. Specifically, the function takes parameter --first-- and composes it with parameter --second-- to calculate and return the composed value. A typical example of the usage of this function is to determine the goal position of a configuration in global coordinates. In such an example, the first argument would be the original configuration and the second argument would be the goal configuration in the original configuration's local coordinate system. The returned value would then be the goal configuration in global coordinates.

37. Circular arc

Syntax: CONFIGURATION *circularArc(double l, double alpha)

Description: Given alpha and the arc length l, this function calculates the final configuration (see figure below). The main purpose of this function is to be used in conjunction with compose function to form the new localization function. In this case, length would actually be delta-s and alpha would be delta-theta. CircularArc() would determine the configuration after the incremental move in the local coordinate system of the original configuration. Then compose() would then take the original configuration (in

global coordinates) and the incremental configuration (in local coordinates) to determine the incremental configuration in global coordinates.

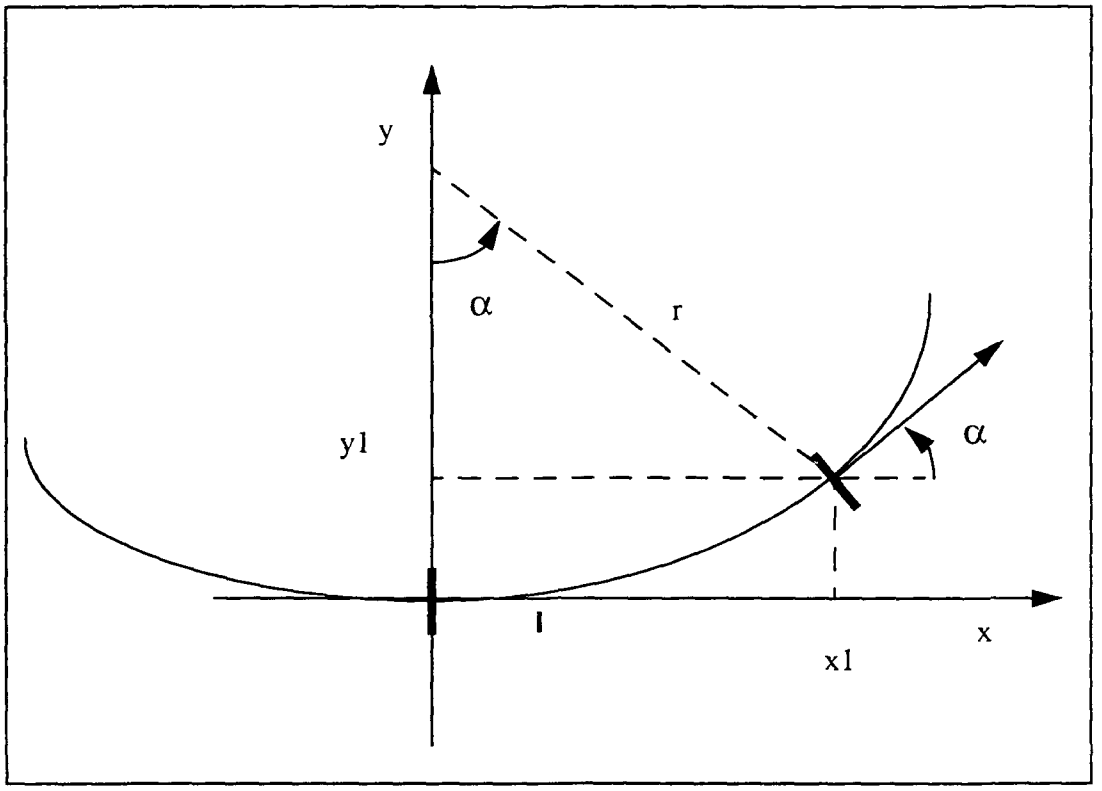


Figure 14: Circular arc after (Kanayama, 94)

B. SYSTEM FUNCTIONS

1. Initialize the Motion Control System

Syntax: void InitMotion(void)

Description:

This function initializes the motion control system. Specifically, InitMotion() initializes all of the variables local to motion control module to their default values. Also in this initializing function, SetMotionIntMechanism() is called which establishes the synchronous interrupt mechanism for motion control. Next, the wheels subsystem is

initialized with the InitalizeWheels() function call. Finally, the wheels of the robot are enabled withe the MotionOn() function call.

2. Motion Control Interrupt

Syntax: void MotionSysControl(void)

Description:

Function MotionSysControl(void) is the interrupt handler workhorse and is called from the assembly interrupt handler shell. Its first task is to update the change in position and orientation through calls to the module responsible for movement. It then uses this information in the motion control laws to derive the commanded linear and rotational velocities required for this motion control cycle. Finally, it passes these computed velocities back to the movement module for execution.

3. Set Path Element

syntax: void setPathElement(PATH_ELEMENT newPath)

Description:

The function setPathElement() is one of several functions that act as an interface for modules outside of MotionSysControl() to access variables that are within MotionSysControl(). Specifically, setPathElement() sets the value of the current path element in motion control to the path element passed in as a parameter.

4. Get Path Element

Syntax: PATH_ELEMENT getPathElement(void)

Description:

Similar to setPathElement, getPathElement acts an interface for functions outside of the motion control module. The function returns the current path element in the motion control module.

5. Get Current Image

Syntax: CONFIGURATION getCurrentImage(void)

Description:

This function is also an interface for functions outside of the motion control module. This function returns that current image that is in the motion control module.

6. Initialize Motion Support Commands

Syntax: void InitMotionSpt(void)

Description:

This function initializes the variables used in motionsupport.c.

7. Update Total Distance

Syntax: void updateTotalDistance(double deltaDistance)

Description:

This function adds the value of the parameter to the running total distance.

8. Get Parabola Configuration(Immediately)

Syntax: PARA getParabolaImm(void)

Description:

This function retrieves the latest parabola that has been processed by the robot. It was developed for the paraRule() function in motion control.

9. Motion On

Syntax: void MotionOn(void)

Description:

This function enables the wheels on the robot. It is called in InitMotion().

10. Motion Off

Syntax: void MotionOff(void)

Description:

This function disables the wheels on the robot. It is called in main.c after user().

11. Blink the LED

Syntax: void blinkLED(void)

Description:

This function controls the output from the interrupt driven motion control system. Specifically, this function turns a LED on for a second and off for a second, repeatedly while the motion control interrupt is being called. This is helpful for debugging purposes. If the light stops blinking, you know that the motion control system is not being called.

12. Limit Robot Movement

Syntax: double limit(double ystar)

Description:

This function is used by the steering function to keep the robot from doing loops when the distance between the robot's position and the path that the robot is following is very large.

VII. RESULTS

With the new version of MML, many significant improvements are achieved. First, global variables are reduced from 152 to zero. Using the global variable and call hierarchy trace, we limit all variable visibility to a module or function level scope. This directly enhances coupling because dependencies between modules caused by global variables is eliminated. Any function from an outside module has to access these variables through function interfaces. Also, data flow is further controlled by statically declaring variables and functions. Modifiability is improved because a developer needs only to be concerned with variables in the module he or she is working in. Finally, functional cohesion is improved because data are encapsulated and functions are structured logically. For example, in MML10, there are several occasions where code that supports a motion command is found in various modules. One in particular is the rotate command. Some of its code is in the motion control module and the rest is in the instruction buffer module. In MML11, all of the rotate command related code in located in the motion control module.

Recent work on the new MML provide testimony to MML improvements. For example, members of the Yamabico group read and understood the new code without having the implementor available. They attribute this high degree of understanding and readability to the extensively documented code in the form of comments. Also, members have already began adding new functionality to the new MML. For example, one member is adding parabola tracking logic to motion control. Instead of having to study the entire motion control module, we simply define the interfaces needed. He is only concerned with the interfaces--receiving the information he needs for his function, and providing the data needed in the calling path tracking function. Finally, the new MML is stable. It runs correctly consistently, and does what is expected. Furthermore, since little or no adjustments to already tested code is necessary, new functions and modules are running quickly with fewer initial run-time errors.

A major focus throughout the design and implementation of MML11 has been the grouping of data. As a result, most data is encapsulated with the same module as the functions that use them. For example, motion control related data is located with motion control related functions in the motion control module. This is true throughout the entire Yamabico subsystems [illegible] evolved, as a consequence, is an object oriented designed system. Although ANSI [illegible] not an Object Oriented Language (OOL), our design has followed this methodology through the grouping of data. This makes it easy to move MML to an OOL in its next upgrade.

VIII. CONCLUSIONS

A. SUMMARY

Our initial goal was to construct a software system for a real-time mobile robot that was stable, readable, and easily modifiable. We first examined those attributes that were desirable in any software system. Then we analyzed MML10 using global variable and call-hierarchy tracing. Considering the desired software attributes, the structural challenges of MML10, and the design assumptions for the MML end users, we developed design goals for MML11. Following from the results of the previous chapter, we achieved our goal. The new system was more stable, easier to read and understand, and simpler to modify.

B. FUTURE RESEARCH

There are two areas of recommend future research. First, a graphical simulator for Yamabico based on MML11, would be a possible area of research. Considering the modular design of MML, this would not be a monumental task.

With the completion of this thesis, MML is now completely written in ANSI C. In restructuring the code, data has been encapsulated providing a object designed software system. Yamabico lends itself to this type of design methodology, because its subsystems, such as wheels, sonar, etc., provide a modular structure in which to design the software. The next logical step would be to write MML in an Object Oriented Language (OOL) such as C++.

APPENDIX A. MOTION CONTROL RELATED MODULES

A. DEFINITIONS.H

```
/*********************************************************************
  File Name:  definitions.h
  Description: This file contains standard definitions and data type
          declarations used throughout the rest of the MML system.
*********************************************************************/

#ifndef __DEFINITIONS_H
#define __DEFINITIONS_H

/* Always include this because it is needed by most modules */
#include "constants.h"

typedef unsigned char  BYTE;
typedef unsigned short WORD;
typedef unsigned long  LONG;
typedef unsigned long *ADDRESS;

typedef enum MODE {NOMODE, STOPMODE, PATHMODE, ROTATEMODE,
            PARAMODE, BIDIRMODE, KSPIRALMODE} MODE;
typedef enum CLASS {NOCLASS, LINECLASS, CIRCLECLASS, BLINECLASS} CLASS;

typedef enum {FALSE = 0, TRUE} BOOLEAN;

typedef struct {
 MODE    mode;
 CLASS   class;
} PATH_TYPE;

typedef struct {
  double  Linear;
  double  Rotational;
} VELOCITY;

typedef struct {
  double  X;
  double  Y;
} POINT;

typedef struct{
  POINT   Posit;
  double  Theta;
  double  Kappa;
} CONFIGURATION;

typedef struct{
 POINT       Focus;
```

```
  CONFIGURATION  Directrix;
} PARA;

typedef struct{
  CONFIGURATION  config;
  PATH_TYPE      pathType;
} PATH_ELEMENT;

#endif
```

B. MAIN.C

```
/*****************************************************************************
  File Name:  main.c
  Description: This file contains main(). Its purpose is to initialize all
          sub-systems and then pass control to user(). Once user() is
          complete, the routine returns control to the resident debugger.
*****************************************************************************/

#include "definitions.h"
#include "memsys.h"
#include "serial.h"
#include "queue.h"
#include "trace.h"
#include "motion.h"
#include "time.h"
#include "sonar.h"
#include "system.h"


/*** Local Prototypes ***/
void user(void);
void __main(void);
void Unexpected(void);



int
main()
{
  InitCPU();

  InitTime();

  InitQueue();          /* init instruction buffer */

  InitTrace();          /* init trace mechanism */

  InitMemsys();    /* memory manager - lites LED 5 */
```

```
  InitMotion();          /* init motion, wheels. and motion logging */

#ifdef SONAR
  InitSonar();
#endif

  DisableInterrupts();

    /* All functions above here must initialize variables only. They
       should not rely on any interrupt handlers, timers, etc. */

  InitHardware();        /* init interrupt handlers and HW registers */
                    /* Handles ALL hardware including motion,
                       serial and sonar */

#ifdef TIMER
    /* fineTiming is used for timing the motion control cycle */
  InitClocktick(0);
#endif

  EnableInterrupts();

  MotionOn();

  user();

  MotionOff();

  IOclose();    /* dump all the data files to the host */

  rexit();      /* clean-up */

  return 0;
}



/*****************************************************************************
  Routine ___main is required when using the 'gcc' compiler. This is because the
  compiler inserts a call to this routine at the beginning of the main function
  defined for the program. This is normally taken care of by linking in the
  bootstrap object modules, however these are not added to a program that
  operates without an operating system such as the mml program. Therefore, since
  this routine is called, the only requirement is for this routine to simply
  return back to the main program.
*****************************************************************************/

void __main()
{ /* empty */ }
```

```
/*************************************************************************
Function Unexpected is the C version of Scott's blank interrupt handler
*************************************************************************/

void Unexpected(void)
{ /* empty */ }
```

C. MOTION.H

```
/*************************************************************************
  File Name:  motion.h
  Description: This file contains the prototypes/interface for the functions
available in the motion
 control module.
 *************************************************************************/

#ifndef __MOTION_H
#define __MOTION_H

#include "definitions.h"


/*************************************************************************
  Function: InitMotion()
  Purpose:  initializes the motion subsystem by assigning default values to the local variables
            and establishing the interrupthandling mechanism.
 Parameters: none
 Returns: void
 Comments:
*************************************************************************/
void    InitMotion(void);


/*************************************************************************
  Function:  MotionSysControl()
  Purpose: the interrupt handler workhorse and is called from the assembly interrupt
            handler shell.
  Parameter: none
  Returns: void
 Comments:
*************************************************************************/
void    MotionSysControl(void);


/*************************************************************************
  INTERFACE  FUNCTIONS SECTION

  The following section defines the functions that provde an interface to values
  within in the motion control modules by functions in other modules.  These
  routines are public.
```

```
**********************************************************************/


/**********************************************************************
 Function: setPathElement()
 Purpose: Sets the value of the current path element in motion control to
       the path element passed in as a parameter.
 Parameters: PATH_ELEMENT newPath
 Returns: void
 Comments:
**********************************************************************/
void    setPathElement(PATH_ELEMENT newPath);


/**********************************************************************
 Function: getPathElement()
 Purpose: retrieves the current path element in the motion control module
 Parameters: void
 Returns: PATH_ELEMENT
 Comments:
**********************************************************************/
PATH_ELEMENT    getPathElement(void);


/**********************************************************************
 Function: getCurrentImage()
 Purpose: retrieves the current image that is in the motion control module
 Parameters: void
 Returns: CONFIGURATION -- the current image
 Comments.
**********************************************************************/
CONFIGURATION    getCurrentImage(void);


/**********************************************************************
 Function: setRobotConfigImm()
 Purpose: To set and update the robot configurations
 Parameters: CONFIGURATION NewConfig
 Returns: void
 Comments:
**********************************************************************/
void          setRobotConfigImm(CONFIGURATION NewConfig);


/**********************************************************************
 Function: getRobotConfig()
 Purpose: Retreives the current robot configuration
 Parameters: Pointer the a variable where the current values for the robot's
       configuration will be placed.
 Returns: void
```

```
 Comments:
*****************************************************************************/
CONFIGURATION    getRobotConfig(void);


/*****************************************************************************
 Function: setConfigImm()
 Purpose:  To set and update the robot's position and thata but not
           it's kappa
 Parameters: CONFIGURATION NewConfig
 Returns:  void
 Comments:
*****************************************************************************/
void          setConfigImm(CONFIGURATION NewConfig);

#endif
```

D. MOTION.C

```
/*****************************************************************************

  File Name:   motion.c
   Description: This file provides the routines and data structures needed to
provide the motion control capability for the robot.
 *****************************************************************************/

#include "definitions.h"
#include "wheels.h"
#include "math.h"
#include "queue.h"
#include "motionlog.h"
#include "geometry.h"
#include "iosys.h"
#include "stdiosys.h"
#include "time.h"
#include "system.h"
#include "trace.h"
#include "motion.h"
#include "motionsupport.h"
#include "seqcmd.h"

/*****************************************************************************

  PRIVATE SECTION

  The following section defines the encapsulated definitions, data structures
  and prototypes used in the system.

*****************************************************************************/

/* constant values */
```

```
#define SMALLERROR          0.0001
#define DEFAULT_LIN_ACC     10.0
#define DEFAULT_ROT_ACC     0.5
#define DEFAULT_GOAL_VEL_LIN 20.0
#define DEFAULT_GOAL_VEL_ROT 0.0


static double    aa,bb,cc,kk;  /* used for steering function in PathRule() */

static VELOCITY   Commanded;  /*commanded velocities */

static double    kappaCommanded; /* commanded kappa */

static CONFIGURATION vehicle,
            currentImage; /* local variables that hold the
                        vehicle and current image
                        values during motion control
                        cycles */

static PATH_ELEMENT  currentPath; /* holds the current path element values */


#ifdef DEBUG
#define  DEBUG_FILE      "debug.log"
#define  DEBUG_SIZE      0x10000
#define  DEBUG_FREQUENCY  1

IOhandle    debugIO;
#endif


#ifdef TIMER
#include "clocktick.h"

#define  TIMER_FILE      "timer.log"
#define  TIMER_SIZE      0x10000
#define  TIMER_FREQUENCY  1

IOhandle TimerLog;
#endif



/*****************************************************************************
  The following static function declarations are the prototypes for the
  encapsulated functions.
*****************************************************************************/

/* calculates the vehicle's next configuration based on the distance
 travelled in the last motion control cycle */
static CONFIGURATION
```

```
localize(CONFIGURATION robot, double deltaS, double deltaTheta);


/* calculates the next commanded linear velocity value. */
static double GetLinearVelocity(double ActualVelocity,
                                double LastCommandedVelocity);


/* calculates the distance remaining on a path to reach a configuration.
   Used with bline calculation. */
static double restOfPath(void);


/*  determines whether the vehicle needs to decelerate.
    Used in bline calculation */
static int needsToDecelerate(double actualVelocity);


/* determines whether it's time to process the next instruction */
static void transition(void);


/* calles a motion rule function based on the mode of travel that the
   vehicle is in */
static VELOCITY motionRules(VELOCITY Actual, VELOCITY Commanded);


/* motion rule for following a path */
static VELOCITY pathRule(VELOCITY Actual, VELOCITY Commanded);


/* motion rule for stopping */
static VELOCITY stopRule(VELOCITY Actual, VELOCITY Commanded);


/* motion rule for rotating */
static VELOCITY rotateRule(VELOCITY Actual, VELOCITY Commanded);


/* motion rule for following a K-spiral */
static VELOCITY spiralRule(VELOCITY Actual, VELOCITY Commanded);


/* determines the Y-star for a vehicle following a line */
static double  computeLineYstar(void);


/* determines the Y-star for a vehicle following a circle */
static double computeCircleYstar(void);
```

```
/* updates the vehicle image when it is following a line */
static void updateLineImage(void);


/* updates the vehicle image when it is following a circle */
static void updateCircleImage(void);

/*****************************************************************************

  MOTION CONTROL SECTION

  The following section defines the functions that provide access to the
  motion control system. These routines are public.

*****************************************************************************/

/*****************************************************************************
 Function InitMotion() initializes all of the private global variables
  in this module to the default values. It then calls SetTimer to
  program the 5th timer on serial board #1(the second serial board)to generate
  synchronous interrupts every 10ms. After the timer has been set up, the
  interrupt handling routine is made available to the system by the call to
  SetMotionInterruptHandler().
*****************************************************************************/

void
InitMotion(void)
{

/* Initialize motion related systems */
  InitMotionsupport();
  InitSeqcmd();
  InitWheels();

/* Initializes the distance. Updated every motion control cycle by deltaS */
  setTotalDistanceImm(0.0);

/* Initialize the goal velocities */
  setGoalLinVelImm(DEFAULT_GOAL_VEL_LIN);
  setGoalRotVelImm(DEFAULT_GOAL_VEL_ROT);

/* Initialize the commanded velocities */
  Commanded.Linear = 0.0;
  Commanded.Rotational = 0.0;

/* Initalize the linear and rotational acceleration */
  setGoalLinAccImm(DEFAULT_LIN_ACC);
  setGoalRotAccImm(DEFAULT_ROT_ACC);

/* Initialize the size constant */
  setSizeConstantImm(DIST_CONSTANT);
```

```
/* Initialize the commanded kappa  */
  kappaCommanded = 0.0;

/* Initialize the vehicle configuration */
  vehicle.Posit.X = 0.0;
  vehicle.Posit.Y = 0.0;
  vehicle.Theta = 0.0;
  vehicle.Kappa = 0.0;

/* Initalize the current path configuration */
  currentPath.config.Posit.X = 0.0;
  currentPath.config.Posit.Y = 0.0;
  currentPath.config.Posit.Y = 0.0;
  currentPath.config.Theta = 0.0;
  currentPath.config.Kappa = 0.0;
  currentPath.pathType.mode = STOPMODE;
  currentPath.pathType.class = NOCLASS;

/* The following 4  variables are used in the steering function found
  in the pathRule() which is in this module  */

  kk = 1.0 / getSizeConstant();
  aa = 3.0 * kk;
  bb = 3.0 * kk * kk;
  cc = kk * kk * kk;

/*  enables the wheels.  Is turned off at the end of main.c after
  user() is called  */

/*********************************************/
/* Initialize data logging here if necessary */
/*********************************************/

#ifdef DEBUG
  debugIO = IOopen(DEBUG_FILE, DEBUG_SIZE, DEBUG_FREQUENCY);
#endif


#ifdef TIMER
  TimerLog = IOopen(TIMER_FILE, TIMER_SIZE, TIMER_FREQUENCY);
#endif

}



/*****************************************************************************
  Function: MotionSysControl()
  Purpose:  the interrupt handler workhorse
  Parameters:  None
```

```
  Return: void
  Comments:It is called from the assembly interrupt handler shell. Its first task is
   to update the change in position and orientation through calls to the module
   responsible for     movement. It then uses this information in the motion control
   laws to derive the commanded linear and rotational velocities required for this
   motion control cycle. Finally, it passes these computed velocities back to
   the movement module for execution.
*****************************************************************************/

void
MotionSysControl(void)
{
  double deltaTheta;
  double deltaS;
  VELOCITY Actual; /* variable used to hold the actual vehicle velocity */

#ifdef TIMER
  int  tick = getCount();
#endif

/* updates the distance traveled by both wheels--found  wheels.c */
  UpdateMovement();

/*returns the linear distance the vehicle  has travelled between the last two
calls to UpdateMovement()--found in wheels.c */

  deltaS = GetDistanceTraveled();

/* returns the difference between the  changes in the distance of the left
and right wheels between the  last two calls to UpdateMovement(). Found in wheels.c */

  deltaTheta = GetOrientationChange();

/* Keeps track of the total distance traved by vehicle */

  updateTotalDistance(deltaS);

  /*update the vehicle's configuration based on the distance travelled
    during the last motion control cycle */;

  vehicle = localize(vehicle, deltaS, deltaTheta);

  /* next 2 lines calculate the actual velocity that robot maintained based
     on the distance travelled over the last motion control cycle. */

  Actual.Linear = deltaS / MOTION_CONTROL_CYCLE;
  Actual.Rotational = deltaTheta / MOTION_CONTROL_CYCLE;

  /* logs the values of the vehicle configuration. Data is
     written to a buffer during each motion control cycle and
     then downloaded to a file when the program ends.  LogMotion
```

```
    is found in motionlog.c */

  LogMotion(vehicle);

    /* This can be relocated just about anywhere... */
#ifdef DEBUG
  IOprintf(debugIO, " ");
#endif

  /* motionRules returns the commanded velocities that will be
    used in the next motion control cycle.  Found in this module.*/

  Commanded = motionRules(Actual,Commanded);

  /* SetMovement() translates the commanded linear and
    rotational velocities into commanded velocities for each
    wheel.  Found in wheels.c */

  SetMovement(Commanded.Linear,Commanded.Rotational);

  transition(); /* reads next instruction if needed. Found in this module.*/

  /* Increments the "time" every motion control cycle for the
    various timer functions.  Found in time.c */

  clockTick();

/* blinkLED is  used to control output from interrupt driven
  motion control system. It turns an LED on and off every
  second. Function found in this module.*/

  blinkLED();

#ifdef TIMER
  IOprintf(TimerLog, "%f \n", (tick - getCount()) / 250.0);
#endif

}


/*****************************************************************************
  Function: localize()
  Purpose:  Calculates the next configuration of the vehicle based on the
        distance that the robot travelled during the last motion
        control cycle
  Parameters:  CONFIGURATION robot --from the last motion control cycle
          double deltaS -- linear distance travelled in last
                  motion control cycle
          double deltaTheta --angular change in the last motion
                  control cycle
  Returns:  CONFIFURATION --of the vehicle based on the distance travlled
```

```
                    during the last motion control cycle
 Comments:
*****************************************************************************/
CONFIGURATION
localize(CONFIGURATION robot, double deltaS, double deltaTheta)
{
CONFIGURATION tempRobot;

  tempRobot = circularArc(deltaS, deltaTheta);
  robot = compose(&robot, &tempRobot);
  robot.Kappa = kappaCommanded;

return robot;
}


/*****************************************************************************
 Function GetLinearVelocity()calculates the linear component of the commanded
  velocity.
*****************************************************************************/

static double
GetLinearVelocity(double ActualVelocity, double CommandedVelocity)
{
  double stopDistance;
  double deceleration;
  double VelocityChange;

  if (currentPath.pathType.class == BLINECLASS &&
      needsToDecelerate(ActualVelocity))
  {
    stopDistance = restOfPath();
    if (stopDistance <= 0.0) {
      CommandedVelocity = 0;
      currentPath.pathType.mode = STOPMODE;
    }
    else {
      deceleration = (ActualVelocity * ActualVelocity)/(2 * stopDistance);
      CommandedVelocity = Max(CommandedVelocity - deceleration *
                    MOTION_CONTROL_CYCLE, 0);
    }
  }
  else {
    VelocityChange = getGoalLinAcc() * MOTION_CONTROL_CYCLE;

    if (ActualVelocity < getGoalLinVel())
      CommandedVelocity = Min(CommandedVelocity + VelocityChange,
                    getGoalLinVel());
    else
      CommandedVelocity = Max(CommandedVelocity - VelocityChange,
                    getGoalLinVel());
```

```
  }

  return CommandedVelocity;
}


/*****************************************************************************
  Function restOfPath() calculates the remaining distance to the ending
  configuration for the BLINE class
*****************************************************************************/
static double
restOfPath(void)
{
  return ((currentPath.config.Posit.X -
         currentImage.Posit.X) * cos(currentImage.Theta) +
        (currentPath.config.Posit.Y -
         currentImage.Posit.Y) * sin(currentImage.Theta));
}


/*****************************************************************************
  Function: needToDecelerate()
Purpose:  To determine whether the robot needs to begin decelerating.  Such
          as in a bline function.
  Parameters:  double actualVelocity (linear)
Returns:  It returns 1 if it needs to decelerate.  Otherwise, it returns 0.
Comments:
*****************************************************************************/
static int
needsToDecelerate(double actualVelocity)
{
   double   decelerate = 0.0;

   if (currentPath.pathType.class == BLINECLASS)  {
     if (2.0 * getGoalLinAcc() * restOfPath() <= actualVelocity * actualVelocity)
       decelerate = 1;
   }

  return decelerate;
}
/*****************************************************************************
  Function:  transition()
  Purpose:  If the leaving point flag is true then read the next instruction
  Parameters:
  Returns:  void
  Comments:
*****************************************************************************/
static  void
transition()
{
   switch(currentPath.pathType.mode)  {
```

```
    case STOPMODE:
      ProcessInstruction();
      break;

    case PATHMODE:
      switch(currentPath.pathType.class) {
        case LINECLASS:
          if (isAtTransitionPt())
            ProcessInstruction();
          break;

        case BLINECLASS:
          break;

        case NOCLASS:
        case CIRCLECLASS:
        default:
          break;
      } /* class switch */
      break;

    case NOMODE:
    case ROTATEMODE:
    case PARAMODE:
    case BIDIRMODE:
    case KSPIRALMODE:
    default:
      break;
  } /* mode switch */
}

/*****************************************************************************
 Function: motionRules()
 Purpose: To calculate the linear velocity and rotational velocity based
        on the type of motion that the robot is executing.
 Parameters: VELOCITY actual, commanded
 Returns: The commanded linear and rotational velocities,
 Comments:
*****************************************************************************/
static VELOCITY
motionRules(VELOCITY actual, VELOCITY commanded)
{
  switch(currentPath.pathType.mode) {
    case STOPMODE:
      commanded = stopRule(actual,commanded);
      break;

    case PATHMODE:
      commanded = pathRule(actual,commanded);
      break;
```

```
    case ROTATEMODE:
      commanded = rotateRule(actual,commanded);
      break;

    case KSPIRALMODE:
      commanded = spiralRule(actual,commanded);
      break;

    case NOMODE:
    case PARAMODE:
    case BIDIRMODE:
    default:
      break;
  }
  return commanded;
}


/*****************************************************************************
  Function: pathRule()
  Purpose: To determine the linear and rotational velocities needed to put or
        keep Yamabico on the path.
  Parameters: VELOCITY actual, commanded
  Returns: The required linear velocity, rotational velocity
  Comments:
*****************************************************************************/
static VELOCITY
pathRule(VELOCITY actual, VELOCITY commanded)
{
  double ystar,dkappa, deltaDistance;

  switch(currentPath.pathType.class) {
    case BLINECLASS:
    case LINECLASS:
      ystar = computeLineYstar();
      updateLineImage();
      break;

    case CIRCLECLASS:
      ystar = computeCircleYstar();
      updateCircleImage();
      break;

    case NOCLASS:
    default:
      break;
  }

  dkappa = -aa * (vehicle.Kappa - currentImage.Kappa)
        -bb * norm(vehicle.Theta - currentImage.Theta)
        -cc * limit(ystar);
```

```
  deltaDistance = MOTION_CONTROL_CYCLE * commanded.Linear;
  kappaCommanded = vehicle.Kappa + dkappa * deltaDistance;


  commanded.Linear = GetLinearVelocity(actual.Linear,commanded.Linear);
  commanded.Rotational = kappaCommanded * commanded.Linear;

  return commanded;
}


/*****************************************************************************
 Function: stopRule()
 Purpose:  updates the commanded velocity to 0 (zero) to stop the robot
 Parameters: VELOCITY actual, commanded
 Returns: The required linear velocity, rotational velocity
 Comments:
*****************************************************************************/
static VELOCITY
stopRule(VELOCITY actual, VELOCITY commanded)
{
  commanded.Linear = 0.0;
  commanded.Rotational = 0.0;
  return commanded;
}


/*****************************************************************************
 Function: rotateRule()
 Purpose:  updates the commanded velocity to rotate the robot
 Parameters: VELOCITY actual, commanded
 Returns: The required linear velocity, rotational velocity
 Comments:
*****************************************************************************/
static VELOCITY
rotateRule(VELOCITY actual, VELOCITY commanded)
{
  return commanded;
}


/*****************************************************************************
 Function: spiralRule()
 Purpose:  To determine the linear and rotational velocities needed to put or
       keep Yamabico on the path.
 Parameters: VELOCITY actual, commanded
 Returns: The required linear velocity, rotational velocity
 Comments:
*****************************************************************************/
static VELOCITY
```

```
spiralRule(VELOCITY actual, VELOCITY commanded)
{
  return commanded;
}


/*****************************************************************************
  Function: computeLineYstar()
  Purpose:  To determine the y* when the robot is tracking a line
  Parameters: none
  Returns: double
  Comments:
*****************************************************************************/
static  double
computeLineYstar()
{
  double ystar;
  CONFIGURATION path = currentPath.config;

  ystar = -(vehicle.Posit.X - path.Posit.X) *
        sin(path.Theta) +
        (vehicle.Posit.Y - path.Posit.Y) *
        cos(path.Theta);
  return ystar;
}


/*****************************************************************************
  Function: computeCircleYstar()
  Purpose:  To determine the y* when the robot is tracking a line
  Parameters: none
  Returns: double
  Comments:
*****************************************************************************/
static  double
computeCircleYstar()
{
  /*  not implemented yet  */
   return 0.0;
}


/*****************************************************************************
  Function:  updateLineImage()
  Purpose:   To update the current image of the vehicle tracking a line
  Parameters: none
  Returns:   void
  Comments:
*****************************************************************************/
static  void
updateLineImage(void)
```

```
{
  double       closest;
  CONFIGURATION  path = currentPath.config;

  closest =(((vehicle.Posit.Y - path.Posit.Y) * cos(path.Theta)) -
        ((vehicle.Posit.X - path.Posit.X) * sin(path.Theta)));

  currentImage.Posit.X = vehicle.Posit.X + closest * sin(path.Theta);
  currentImage.Posit.Y = vehicle.Posit.Y - closest * cos(path.Theta);
  currentImage.Theta = path.Theta;
  currentImage.Kappa = path.Kappa;

  return;
}

/*****************************************************************************
 Function: updateCircleImage()
 Purpose:  To compute the current image of the vehicle tracking a circle
 Parameters: none
 Returns:  void
 Comments:
*****************************************************************************/
static void
updateCircleImage(void)
{
  double       gamma, radius;
  POINT        origin;
  CONFIGURATION  path = currentPath.config;

  radius = (1.0 / path.Kappa);

  origin.X = path.Posit.X - radius * sin(path.Theta);
  origin.Y = path.Posit.Y + radius * cos(path.Theta);

  gamma = atan2(vehicle.Posit.Y - origin.Y, vehicle.Posit.X - origin.X);

  currentImage.Posit.X = origin.X + fabs(radius) * cos(gamma);
  currentImage.Posit.Y = origin.Y + fabs(radius) * sin(gamma);
  currentImage.Theta = norm(gamma + (PI/2) * (path.Kappa/fabs(path.Kappa)));
  currentImage.Kappa = path.Kappa;
}




/*****************************************************************************
  INTERFACE  FUNCTIONS SECTION

  The following section defines the functions that provde an interface to values
  within in the motion control modules by functions in other modules.  These
```

```
  routines are public.

*****************************************************************************/


/*****************************************************************************
 Function: setPathElement()
 Purpose: Sets the value of the current path element in motion control to
        the path element passed in as a parameter.
 Parameters: PATH_ELEMENT newPath
 Returns: void
 Comments:
*****************************************************************************/
void
setPathElement(PATH_ELEMENT newPath)
{
  DisableInterrupts();

  currentPath.config.Posit.X = newPath.config.Posit.X;
  currentPath.config.Posit.Y = newPath.config.Posit.Y;
  currentPath.config.Theta = newPath.config.Theta;
  currentPath.config.Kappa = newPath.config.Kappa;
  currentPath.pathType.mode = newPath.pathType.mode;
  currentPath.pathType.class = newPath.pathType.class;

  EnableInterrupts();
}


/*****************************************************************************
 Function: getPathElement()
 Purpose: retrieves the current path element in the motion control module
 Parameters: void
 Returns: PATH_ELEMENT
 Comments:
*****************************************************************************/
PATH_ELEMENT
getPathElement(void)
{
  return currentPath;
}


/*****************************************************************************
 Function: getCurrentImage()
 Purpose: retrieves the current image that is in the motion control module
 Parameters: void
 Returns: CONFIGURATION -- the current image
 Comments:
*****************************************************************************/
CONFIGURATION
```

```
getCurrentImage(void)
{
  return currentImage;
}


/*****************************************                    ***********************
 Function: setRobotConfigImm()
 Purpose: To set and update the robot configuration
 Parameters: CONFIGURATION NewConfig
 Returns: void
 Comments:
*****************************************************************************/
void
setRobotConfigImm(CONFIGURATION NewConfig)
{
  DisableInterrupts();

  vehicle.Posit.X = NewConfig.Posit.X;
  vehicle.Posit.Y = NewConfig.Posit.Y;
  vehicle.Theta = NewConfig.Theta;
  vehicle.Kappa = NewConfig.Kappa;

  EnableInterrupts();
}


/*****************************************************************************
 Function: getRobotConfig()
 Purpose: Retreives the current robot configuration
 Parameters: Pointer the a variable where the current values for the robot's
        configuration will be placed.
 Returns: void
 Comments:
*****************************************************************************/
CONFIGURATION
getRobotConfig(void)
{
  DisableInterrupts();
  return vehicle;
  EnableInterrupts();
}


/*****************************************************************************
 Function: setConfigImm()
 Purpose: To set and update the robot's position and thata but not
       it's kappa
 Parameters: CONFIGURATION NewConfig
 Returns: void
 Comments:
```

```
*****************************************************************************/
void
setConfigImm(CONFIGURATION NewConfig)
{
  DisableInterrupts();

  vehicle.Posit.X = NewConfig.Posit.X;
  vehicle.Posit.Y = NewConfig.Posit.Y;
  vehicle.Theta = NewConfig.Theta;

  EnableInterrupts();
}
```

E. MOTIONSUPPORT.H

```
/*****************************************************************************
 File name: motionsupport.h
 Description: contains miscellaneous functions prototypes that support
        motion control
 Revision history:
*****************************************************************************/

#ifndef __MOTIONSUPPORT_H
#define __MOTIONSUPPORT_H


/*****************************************************************************
 Function: InitMotionSpt()
 Purpose:  Initializes the variables used in motionsupport.c
 Parameters: void
 Returns:  void
 Comments:
*****************************************************************************/
void    InitMotionsupport(void);


/*****************************************************************************
 Function: stopImm()
 Purpose:  updates the goal velocity to zero inorder to stop the robot
 Parameters:  void
 Returns:  void
 Comments: This is the immediate stop command
*****************************************************************************/
void    stopImm(void);


/*****************************************************************************
 Function:  setGoalLinVelImm()
 Purpose:  sets and updates the goal linear velocity of the robot
 Parameters: double LinearVelocity
```

```
 Returns: void
 Comments:
*****************************************************************************/
void    setGoalLinVelImm(double LinearVelocity);


/*****************************************************************************
 Function: getGoalLinVel()
 Purpose: Retreives the current goal linear velocity
 Parameters: void
 Returns: double
 Comments:
*****************************************************************************/
double  getGoalLinVel(void);


/*****************************************************************************
 Function: setGoalRotVelImm()
 Purpose: Sets and updates the goal rotational velocity
 Parameters: double RotationalVelocity
 Returns: void
 Comments:
*****************************************************************************/
void    setGoalRotVelImm(double RotationalVelocity);


/*****************************************************************************
 Function: getGoalRotVel()
 Purpose: retreivies the current goal rotational velocity
 Parameters: void
 Returns: double
 Comments:
*****************************************************************************/
double    getGoalRotVel(void);


/*****************************************************************************
 Function: setGoalLinAccImm()
 Purpose:  Sets and updates the goal linear acceleration
 Parameters: double LinearAcceleration
 Returns: void
 Comments:
*****************************************************************************/
void    setGoalLinAccImm(double LinearAcceleration);


/*****************************************************************************
 Function: getGoalLinAcc()
 Purpose: retreives the current goal linear acceleration
 Parameters: void
 Returns: double
```

```
 Comments:
*****************************************************************************/
double     getGoalLinAcc(void);


/*****************************************************************************
 Function: setGoalRotAccImm()
 Purpose: Sets and updates the goal rotational acceleration
 Parameters: double RotationalAcceleration
 Returns: void
 Comments:
*****************************************************************************/
void     setGoalRotAccImm(double RotationalAcceleration);


/*****************************************************************************
 Function: getGoalRotAcc()
 Purpose: Retreives the current goal rotational acceleration
 Parameters: void
 Returns: double
 Comments:
*****************************************************************************/
double     getGoalRotAcc(void);


/*****************************************************************************
 Function: setSizeConstantImm()
 Purpose: sets the size constant which influences the sensitivity of the
          steering function
 Parameters: double sizeConstant
 Returns: void
 Comments:
*****************************************************************************/
void     setSizeConstantImm(double SizeConstant);


/*****************************************************************************
 Function: getSizeConstant()
 Purpose: returns the current size constant being used in motion control
 Parameters: void
 Returns: double size constant
 Comments:
*****************************************************************************/
double     getSizeConstant(void);


/*****************************************************************************
 Function: setTotalDistanceImm()
 Purpose: sets the total distance travelled by the rc          e value passed
       as a parameter
 Parameters: double distance
```

```
 Returns: void
 Comments:
*****************************************************************************/
void    setTotalDistanceImm(double distance);


/*****************************************************************************
 Function: updateTotalDistance()
 Purpose: adds the value of the parameter to the running total distance
 Parameters: double deltaDistance
 Returns: void
 Comments:
*****************************************************************************/
void    updateTotalDistance(double deltaDistance);


/*****************************************************************************
 Function: getTotalDistance()
 Purpose: returns the total distance travelled by the robot
 Parameters: void
 Returns: double totalDistance
 Comments:
*****************************************************************************/
double    getTotalDistance(void);


/*****************************************************************************
 Function: getTotalDistanceImm()
 Purpose: returns the total distance travelled by the robot
 Parameters: void
 Returns: double totalDistance
 Comments:
*****************************************************************************/
double    getTotalDistanceImm(void);


/*****************************************************************************
 Function: haltMotionImm()
 Purpose: brings the robot to a rest. Is different from stop in that it's
          original goal velocity is saved to be later used by the resume
          command to allow the robot to continue travelling at the same
          speed.
 Parameters: void
 Returns: void
 Comments:
*****************************************************************************/
void    haltMotionImm(void);


/*****************************************************************************
 Function: resumeMotionImm()
```

```
 Purpose:  Allows the robot to resume the speed it was travelling before the
       haltMotionImm() command was given.
 Parameters: void
 Returns: void
 Comments:
*****************************************************************************/
void    resumeMotionImm(void);


/*****************************************************************************
 Function:  parabolaImm()
 Purpose:  Immediate command that allows the robot to follow the parabola
       passed in the parameter
 Parameters: PARA newParabola
 Returns: void
 Comments:
*****************************************************************************/
void    parabolaImm(PARA newParabola);


/*****************************************************************************
 Function: getParabolaImm()
 Purpose:  retrieves the latest parabola that has been processed by the robot
 Parameters: void
 Returns: PARA parabola
 Comments: this function was developed for the paraRule() function in motion
       control
*****************************************************************************/
PARA    getParabolaImm(void);


/*****************************************************************************
 Function:  MotionOn()
 Purpose:  enables the wheels
 Parameters:void
 Returns: void
 Comments:
*****************************************************************************/
void    MotionOn(void);


/*****************************************************************************
 Function: MotionOff()
 Purpose:  disables the wheels
 Parameters: void
 Returns: void
 Comments:
*****************************************************************************/
void    MotionOff(void);
```

```
/*****************************************************************************
Function: blinkLED()
Purpose: To control the output from the interrupt driven motion control
        system. LoopTest is sassigned zero every second.
 Parameters: void
 Returns: none
 Comments:
*****************************************************************************/
void    blinkLED(void);


/*****************************************************************************
FUNCTION:  limit
PURPOSE:   This function is used by the steering function to keep the robot
        from doing loops when ystar is very large
PARAMETERS: double  ystar (unlimited)
RETURNS:   double ystar (limited)
COMMENTS:  originally written 7 December 92 - Dave MacPherson
    updated for MML11 8 June 94 - Kevin Huggins
*****************************************************************************/
double     limit(double ystar);


/*****************************************************************************
 Function: isAtTransitionPt()
 Purpose: Is true if the transition point has been reached
 Parameters: none
 Returns:  1 or 0
 Comments: Presently a dummy function until the transition point calculation
        module is finished, then it will be moved there.
*****************************************************************************/
int     isAtTransitionPt();


#endif
```

F. MOTIONSUPPORT.C

```
/*****************************************************************************
 File name: motionsupport.c
 Description: contains miscellaneous functions that support motion control
 Revision history:
*****************************************************************************/

#include "definitions.h"
#include "wheels.h"
#include "motionsupport.h"
#include "motion.h"
#include "system.h"
```

```
BOOLEAN         Halted;


static VELOCITY   haltedVel;
static VELOCITY   goalVel,
                  goalAcc;

static PARA    parabola;
static double  desiredSizeConstant,
               totalDistance;
static int     LoopTest,
               testCount;

/*****************************************************************************
  Function: InitMotionSpt()
  Purpose:  Initializes the variables used in motionsupport.c
  Parameters: void
  Returns:  void
  Comments:
*****************************************************************************/

void
InitMotionsupport(void)
{
   LoopTest = 0;
   testCount =  0;
Halted = FALSE;
   totalDistance = 0.0;
   haltedVel.Linear = 0.0;
   haltedVel.Rotational = 0.0;
   parabola.Focus.X = 0.0;
   parabola.Focus.Y = 0.0;
   parabola.Directrix.Posit.X = 0.0;
   parabola.Directrix.Posit.Y = 0.0;
   parabola.Directrix.Theta = 0.0;
   parabola.Directrix.Kappa = 0.0;
}


/*****************************************************************************
  Function: stopImm()
  Purpose:  updates the goal velocity to zero inorder to stop the robot
  Parameters:  void
  Returns:  void
  Comments: This is the immediate stop command
*****************************************************************************/
void
stopImm(void)
{
   WheelsDisable();
```

```
  goalVel.Linear = 0.0;
  goalVel.Rotational = 0.0;
}


/*****************************************************************************
 Function: setGoalLinVelImm()
 Purpose: sets and updates the goal linear velocity of the robot
 Parameters: double LinearVelocity
 Returns: void
 Comments:
*****************************************************************************/
void
setGoalLinVelImm(double linearVelocity)
{
  goalVel.Linear = linearVelocity;
}


/*****************************************************************************
 Function: getGoalLinVel()
 Purpose: Retreives the current goal linear velocity
 Parameters: void
 Returns: double
 Comments:
*****************************************************************************/
double
getGoalLinVel(void)
{
  return goalVel.Linear;
}


/*****************************************************************************
 Function: setGoalRotVelImm()
 Purpose: Sets and updates the goal rotational velocity
 Parameters: double RotationalVelocity
 Returns: void
 Comments:
*****************************************************************************/
void
setGoalRotVelImm(double RotationalVelocity)
{
  goalVel.Rotational = RotationalVelocity;
}

/*****************************************************************************
 Function: getGoalRotVel()
 Purpose: retreivies the current goal rotational velocity
 Parameters: void
 Returns: double
```

```
 Comments:
*****************************************************************************/
double
getGoalRotVel(void)
{
  return goalVel.Rotational;
}

/*****************************************************************************
 Function: setGoalLinAccImm()
 Purpose:  Sets and updates the goal linear acceleration
 Parameters: double LinearAcceleration
 Returns: void
 Comments:
*****************************************************************************/
void
setGoalLinAccImm(double LinearAcceleration)
{
  goalAcc.Linear = LinearAcceleration;
}


/*****************************************************************************
 Function: getGoalLinAcc()
 Purpose: retreives the current goal linear acceleration
 Parameters: void
 Returns: double
 Comments:
*****************************************************************************/
double
getGoalLinAcc(void)
{
  return goalAcc.Linear;
}

/*****************************************************************************
 Function: setGoalRotAccImm()
 Purpose: Sets and updates the goal rotational acceleration
 Parameters: double RotationalAcceleration
 Returns: void
 Comments:
*****************************************************************************/
void
setGoalRotAccImm(double RotationalAcceleration)
{
  goalAcc.Rotational = RotationalAcceleration;
}


/*****************************************************************************
 Function: getGoalRotAcc()
```

```
 Purpose: Retreives the current goal rotational acceleration
 Parameters: void
 Returns: double
 Comments:
*****************************************************************************/
double
getGoalRotAcc(void)
{
  return goalAcc.Rotational;
}

/*****************************************************************************
 Function: setSizeConstantImm()
 Purpose: sets the size constant which influences the sensitivity of the
          steering function
 Parameters: double sizeConstant
 Returns: void
 Comments:
*****************************************************************************/
void
setSizeConstantImm(double sizeConstant)
{
  desiredSizeConstant = sizeConstant;
}


/*****************************************************************************
 Function: getSizeConstant()
 Purpose: returns the current size constant being used in motion control
 Parameters: void
 Returns: double size constant
 Comments:
*****************************************************************************/
double
getSizeConstant(void)
{
  return desiredSizeConstant;
}


/*****************************************************************************
 Function: setTotalDistanceImm()
 Purpose: sets the total distance travelled by the robot to the value passed
       as a parameter
 Parameters: double distance
 Returns: void
 Comments:
*****************************************************************************/
void
setTotalDistanceImm(double distance)
{
```

```
  totalDistance = distance;
}


/*****************************************************************************
  Function: updateTotalDistance()
  Purpose:  adds the value of the parameter to the running total distance
  Parameters:  double deltaDistance
  Returns:  void
  Comments:
*****************************************************************************/
void
updateTotalDistance(double deltaDistance)
{
  totalDistance += deltaDistance;
}


/*****************************************************************************
  Function: getTotalDistance()
  Purpose:  returns the total distance travelled by the robot
  Parameters: void
  Returns: double totalDistance
  Comments:
*****************************************************************************/
double
getTotalDistance(void)
{
  return totalDistance;
}


/*****************************************************************************
  Function: haltMotionImm()
  Purpose:  brings the robot to a rest.  Is different from stop in that it's
        original goal velocity is saved to be later used by the resume
        command to allow the robot to continue travelling at the same
        speed.
  Parameters: void
  Returns: void
  Comments:
*****************************************************************************/
void
haltMotionImm(void)
{
  if (!Halted) {
    Halted = TRUE;
    haltedVel.Linear = goalVel.Linear;
    haltedVel.Rotational = goalVel.Rotational;
    WheelsDisable();
  }
```

```
}

/*****************************************************************************
 Function: resumeMotionImm()
 Purpose:  Allows the robot to resume the speed it was travelling before the
       haltMotionImm() command was given.
 Parameters: void
 Returns: void
 Comments:
*****************************************************************************/
void
resumeMotionImm(void)
{
  if (Halted) {
    Halted = FALSE;
    goalVel.Linear = haltedVel.Linear;
    goalVel.Rotational = haltedVel.Rotational;
    WheelsEnable();
  }
}

/*****************************************************************************
 Function:  parabolaImm()
 Purpose:  Immediate command that allows the robot to follow the parabola
       passed in the parameter
 Parameters: PARA newParabola
 Returns: void
 Comments:
*****************************************************************************/
void
parabolaImm(PARA newParabola)
{
  PATH_ELEMENT pathElement;

  DisableInterrupts();

  pathElement.pathType.mode = PARAMODE;
  setPathElement(pathElement);
  parabola = newParabola;

  EnableInterrupts();
}


/*****************************************************************************
 Function: getParabolaImm()
 Purpose:  retrieves the latest parabola that has been processed by the robot
 Parameters: void
 Returns: PARA parabola
 Comments: this function was developed for the paraRule() function in motion
       control
```

```
*****************************************************************************/

PARA
getParabola( 'm'
{
  return parabola;
}


/*****************************************************************************
 Function:  MotionOn()
 Purpose:  enables the wheels
 Parameters:void
 Returns: void
 Comments:
*****************************************************************************/
void
MotionOn(void)
{
  WheelsEnable();
}



/*****************************************************************************
 Function: MotionOff()
 Purpose:  disables the wheels
 Parameters: void
 Returns: void
 Comments:
*****************************************************************************/
void
MotionOff(void)
{
  WheelsDisable();
}


/*****************************************************************************
Function:  blinkLED()
Purpose:  To control the output from the inperrupt driven motion control
      system.  LoopTest is sassigned zero every second.
 Parameters: void
 Returns: none
 Comments:
*****************************************************************************/
void
blinkLED(void)
{
  if (LoopTest++ >= ( (int)((1.0/MOTION_CONTROL_CYCLE) - 1))) {
    changeLEDstate(7);
```

```
    LoopTest = 0;
  }
}

/*****************************************************************************
FUNCTION:  limit
PURPOSE:   This function is used by the steering function to keep the robot
        from doing loops when ystar is very large
PARAMETERS: double  ystar (unlimited)
RETURNS:   double ystar (limited)
COMMENTS:  originally written 7 December 92 - Dave MacPherson
    updated for MML11 8 June 94 - Kevin Huggins
*****************************************************************************/
double
limit(double ystar)
{
  if(ystar >  2.0 * DIST_CONSTANT)
    return(2.0 * DIST_CONSTANT);
  if (ystar < -2.0 * DIST_CONSTANT)
    return(-2.0 * DIST_CONSTANT);
  return ystar;
}


/*****************************************************************************
 Function:  isAtTransitionPt()
 Purpose:  Is true if the transition point has been reached
 Parameters: none
 Returns:  1 or 0
 Comments: Presently a dummy function until the transition point calculation
        module is finished
*****************************************************************************/
int
isAtTransitionPt()
{
  if (testCount++ >= 600) return 1;
  else return 0;
}
```

G. SEQCMD.H

```
/*****************************************************************************
Module name:  seqcmd.h
 Comments:  has all of the public function prototypes
*****************************************************************************/

#ifndef __SEQCMD_H
#define __SEQCMD_H
```

```
void InitSeqcmd(void);

void line(CONFIGURATION );
void bline(CONFIGURATION );
void stop(void);
void setRobotConfig(CONFIGURATION config);

#endif
```

H. SEQCMD.C

```
/*****************************************************************************
 File name:  seqcmd.c
  Descriptions:  collection of all of the sequential commands that are
                 available to Yamabico
  Revision history:
*****************************************************************************/
#include "definitions.h"
#include "queue.h"
#include "seqcmd.h"
#include "motion.h"
#include "time.h"
#include "iosys.h"
#include "motionsupport.h"

  /* local variables */
static MODE lastMode;

  /* local prototypes */
static int LineProcess(PATH_ELEMENT);
static int BLineProcess(PATH_ELEMENT);
static int SetRobProcess(PATH_ELEMENT);
static int StopProcess(PATH_ELEMENT);




void
InitSeqcmd(void)
{
lastMode = STOPMODE;
}


/*****************************************************************************
  Function:    line configuration function pair
  Purpose:     To read and execute a sequential line command
```

```
**********************************************************************************/
void
line(CONFIGURATION lineConfig)
{
PATH_ELEMENT pathElement;

pathElement.config = lineConfig;
pathElement.pathType.mode = PATHMODE;
pathElement.pathType.class = LINECLASS;

lastMode = PATHMODE;
AddInstruction(pathElement,LineProcess);
}

int
LineProcess(PATH_ELEMENT pathElement)
{
setPathElement(pathElement); /* update the path element in motion.c */
return 1;
}

/**********************************************************************************
Function:  bline configuration function pair
Purpose:   To read and execute a sequential bline command
**********************************************************************************/
void
bline(CONFIGURATION blineConfig)
{
PATH_ELEMENT pathElement;

pathElement.config = blineConfig;
pathElement.pathType.mode = PATHMODE;
pathElement.pathType.class = BLINECLASS;

lastMode = PATHMODE;
AddInstruction(pathElement,BLineProcess);
}

int
BLineProcess(PATH_ELEMENT pathElement)
{
setPathElement(pathElement);
return 1;
}

/**********************************************************************************
Function: stop vehicle function pair
Purpose:  To read and execute a sequential stop command
**********************************************************************************/
void
stop(void)
```

```
{
 PATH_ELEMENT pathElement;

 pathElement.pathType.mode = STOPMODE;

lastMode = STOPMODE;
AddInstruction(pathElement, StopProcess);
}

int
StopProcess(PATH_ELEMENT pathElement)
{
 stopImm();
  setPathElement(pathElement)
  return 1;
}

/*****************************************************************************
Function: set robot configuration function pair
Purpose:  To set the robots' location when it is in a STOP mode
*****************************************************************************/
void
setRobotConfig(CONFIGURATION config)
{
PATH_ELEMENT pathElement;

  if (lastMode != STOPMODE)  {
    /* write some error message */
    return;
  }

pathElement.config = config;
pathElement.pathType.mode = STOPMODE;

lastMode = STOPMODE;
AddInstruction(pathElement, SetRobProcess);
}

int
SetRobProcess(PATH_ELEMENT pathElement)
{
setRobotConfigImm(pathElement.config);
return 0;
}
```

I. USER.C

```
/*****************************************************************************
  File Name:  user.c
  Description: This file contains a sample user program that can be used with
```

```
        the new MML system created using ANSI C.
 ********************************************************************************/

#include "definitions.h"
#include "iosys.h"
#include "stdiosys.h"
#include "serial.h"
#include "motion.h"
#include "sonar.h"
#include "trace.h"
#include "geometry.h"
#include "time.h"
#include "seqcmd.h"
#include "system.h"
#include "immcmd.h"
#include "motionsupport.h"
#include "motionlog.h"

#define ESC 0x1b


/*******************************************************************************
 Function: user()
 Purpose:
 Parameters: void
 Returns:  void
 Comments:  This user program commands the robot to follow a star path.
   It  follows the path offsetting where the robot "thinks" it at by using
   the setConfig() command.
*******************************************************************************/

void
user(void)
{
   int segments = 5;
      CONFIGURATION   start, justGo, currentPosit, jump;
   start = defineConfig(0.0, 0.0, 0.0, 0.0);
   justGo = defineConfig(0.0, 0.1, 0.0, 0.0);
   jump  = defineConfig(0.0, 45.0, -1.5*HPI, 0.0);


      printf(" \12This is the set_config star program.");

  MotionLog("star1.dat", 5, 0);
  setRobotConfig(start);
  line(justGo);
 do{
   waitSec(10);
    currentPosit = getRobotConfig();
    printf("\n current x position is : %f", currentPosit.Posit.X);
    printf("\n current y position is : %f\n", currentPosit.Posit.Y);
```

```
    setRobotConfigImm(compose(&currentPosit,&jump));
    } while(--segments);
}
```

APPENDIX B. GEOMETRIC FUNCTONS

A. GEOMETRY.H

```
/*****************************************************************************
  File Name:  geometry.h
  Description: This file contains the standard geometry functions that are
          called by several functions.
*****************************************************************************/
#ifndef __GEOMETRY_H
#define __GEOMERTY_H

#include "definitions.h"

/*****************************************************************************
 Function: euDis()
 Purpose:  Computes the Eucledian distance between two given points
 Parameters: double x1,y1,x2,y2
 Returns: double
 Comments:
*****************************************************************************/
double        euDis(double x1, double y1,double x2, double y2);


/*****************************************************************************
 FUNCTION:  norm()
 PARAMETERS: double angle    ---- the angle to normalize
 PURPOSE:   normalize the input angle between -PI and PI
 RETURNS:   double:the normalized angle in radians
 COMMENTS:  This is the most common normalizing function used in the system
*****************************************************************************/
double        norm(double angle);


/*****************************************************************************
 FUNCTION:  positiveNorm()
 PARAMETERS: double angle    ---- the angle to normalize
 PURPOSE:   normalize the input angle between 0 and 2PI
 RETURNS:   double:the normalized angle in radians
 COMMENTS:  None
*****************************************************************************/
double        positiveNorm(double angle);
/*****************************************************************************
 FUNCTION:  negativeNorm()
 PARAMETERS: double angle    ---- the angle to normalize
 PURPOSE:   normalize the input angle between -2PI and 0
 RETURNS:   double:the normalized angle in radians
 COMMENTS:  None
*****************************************************************************/
double        negativeNorm(double angle);
```

```
/*****************************************************************************
 FUNCTION:  normPlover2()
 PARAMETERS: double angle     ---- the angle to normalize
 PURPOSE:   normalize the input angle between -PI/2  and PI/2
 RETURNS:   double:  the normalized angle in radians
 COMMENTS:  This was designed specifically for parabola calculations
*****************************************************************************/
double
normPlover2(double angle);

/*****************************************************************************
 FUNCTION: signedDiff()
 PARAMETERS: CONFIGURATIONS directerix
 POINT focus
 PURPOSE: to calculate the size distance between a point and a configuration.
 RETURNS: double: the signed difference
 COMMENTS:
*****************************************************************************/
double
signedDiff(CONFIGURATION config , POINT pt);

/*****************************************************************************
 FUNCTION:  defineConfig()
 PARAMETERS: double x,y,theta,kappa       --The values that define a
                          configuration
 PURPOSE:   To allocate nad assign a configuration
 RETURNS:   CONFIGURATION:a pointer to a configuration
 COMMENTS:  Was called def_configuration() in MML10
*****************************************************************************/
CONFIGURATION    defineConfig(double x,double y,double theta,double kappa);

/*****************************************************************************
 FUNCTION:  defineParabola()
 PARAMETERS: double xf,yf          ---defines the focus
        double xd, yd, thetad  ---defines the directrix
 PURPOSE:   To allocate assign a parabola
 RETURNS:   PARA:a pointer to a parabola type
 COMMENTS:  Was called def_parabola() in MML1υ
*****************************************************************************/
PARA          defineParabola(double xf,double  yf, double xd, double yd, double td);

/*****************************************************************************
 FUNCTION:  reverseOrientation()
 PARAMETERS: CONFIGURATION original  --the original configuration
 PURPOSE:   To reverse the orientation of a given configuration
 RETURNS:   CONFIGURATION: the reversed  configuration
 COMMENTS:  Was called negate() in MML10
*****************************************************************************/
CONFIGURATION    reverseOrientation(CONFIGURATION original);
```

```
/*****************************************************************************
 FUNCTION:  findSymConfig()
 PARAMETERS: double a  -- distance from either point to the intersection of
                  both lines determined by the two configurations
   double alpha --The angular difference between both orientations
 PURPOSE:  This function finds the symmetric configuration of a configuration
        described by alpha and a above.
 RETURNS:  CONFIGURATION: sym_config -- the symmetic configuration
 COMMENTS:  Was called def_sym() in MML10
        One drawback to this function is that it is not possible to
        represent a symmetric configuration whose alpha is equal to PI.
   find_sym_config1() is used to cover these situations
*****************************************************************************/
CONFIGURATION    findSymConfig(double a, double alpha);


/*****************************************************************************
 FUNCTION:  findSymConfig1()
 PARAMETERS: double d  -- distance from the origin (base configuration) to
                  the symmetric configuration.
   double alpha --The angular difference between both orientations
 PURPOSE:  This finds the symmetricconfiguration of a configuration
        described by alpha and a above.
 RETURNS:  CONFIGURATION: sym_config --the symmetic configuration
 COMMENTS:  Was called def_sym1() in MML10
        This function overcomes the drawback of the original
        find_sym_config() of not being able to handle the situation when
        alpha equals PI
*****************************************************************************/
CONFIGURATION    findSymConfig1(double d, double alpha);


/*****************************************************************************

 FUNCTION:  inverse()
 PARAMETERS: CONFIGURATION *original --the original configuration
                  in global coordinates
 PURPOSE:  To calculate the inverse of a given configuration
 RETURNS:  CONFIGURATION: the inversed configuration
              such that;
              original * inversed = Identity
 COMMENTS:  None
*****************************************************************************/
CONFIGURATION    inverse(CONFIGURATION original);



/*****************************************************************************

 FUNCTION:  compose()
```

```
 PARAMETERS: CONFIGURATION *first  -- pointer to the first configuration
                  *second -- pointer to the  second configuration
                  *third  -- pointer to the third configuration
 PURPOSE:   To calculate the composition of the first and second
        configurations
 RETURNS:   CONFIGURATION: pointer to third  configuration which is the
        composition of the first and second configurations
 COMMENTS:  A typical example of the usage of this function is to determine
        the goal position of a configuration in global coordinates.  In
        such an example, the first argument would be the original
        configuration and the second argument would be the goal
        configuration in the original configuration's local coordinate
        system.  The resultant third argument would then be the goal
        configuration in global coordinates. .Was called comp() in MML10
*****************************************************************************/
CONFIGURATION    compose(CONFIGURATION *first,CONFIGURATION *second);


/*****************************************************************************

 FUNCTION:  circularArc()
 PARAMETERS: CONFIGURATION length  --the arc length
                  alpha   --the end orientation
                  config  --pointer to the resultant configuration
 PURPOSE:   Given the arc length and alpha, to calculate the final
        configuration
 RETURNS:   CONFIGURATION: pointer to the final configuration
 COMMENTS:  The main purpose of this functionis to be used in conjunction
        with compose() to fomr a new next(). In this case, length would
        actually be delta-s and alpha would be delta-theta.
CircularArc() would determine the configuration after the incre-
mental move in the local coordinated system of the original
configuration. Then compose() would take the original
configuration (in global coordinates) and the incremental
configuration (in local coordinates) to determine the
incremental configuration in global coordinates.
*****************************************************************************/
CONFIGURATION    circularArc(double length, double alpha);


#endif
```

B. GEOMETRY.C

```
/*****************************************************************************
  File Name:  geometry.c
  Description: This file contains the standard geometry functions that are
          called by several functions.
*****************************************************************************/
```

```
/*#include "math68881.h"*/
#include "definitions.h"
#include "geometry.h"

/*****************************************************************************
 Function: euDis()
 Purpose:  Computes the Eucledian distance between two given points
 Parameters: double x1,y1,x2,y2
 Returns: double
 Comments:
*****************************************************************************/
double
euDis(double x1, double y1,double x2, double y2)
{
 return sqrt((x1 - x2) * (x1 - x2) + (y1 - y2) * (y1 - y2));
}


/*****************************************************************************
 FUNCTION:  norm()
 PARAMETERS: double angle    ---- the angle to normalize
 PURPOSE:   normalize the input angle between -PI and PI
 RETURNS:   double:the normalized angle in radians
 COMMENTS:  This is the most common normalizing function used in the system
        This performs that same as norm() and normalize)() in MML10.
*****************************************************************************/
double
norm(double angle)
{
 while ((angle >= PI) || (angle < -PI))
  {
   if (angle >= PI)
angle -= DPI;
   else
angle += DPI;
  }
 return angle;
}

/*****************************************************************************
 FUNCTION:  positiveNorm()
 PARAMETERS: double angle    ---- the angle to normalize
 PURPOSE:   normalize the input angle between 0 and 2PI
 RETURNS:   double:the normalized angle in radians
 COMMENTS:  Same functionality as pnorm() in MML10
*****************************************************************************/
double
positiveNorm(double angle)
{
 while ((angle >= DPI) || (angle < 0))
  {
```

```
    if (angle >= DPI)
angle -= DPI;
    else
angle += DPI;
   }
 return angle;
}

/*****************************************************************************
 FUNCTION:  negativeNorm()
 PARAMETERS: double angle     ---- the angle to normalize
 PURPOSE:   normalize the input angle between -2PI and 0
 RETURNS:   double:the normalized angle in radians
 COMMENTS:  Same functionality as nnorm() in MML10
*****************************************************************************/
double
negativeNorm(double angle)
{
 while ((angle > 0) || (angle <= -DPI))
   {
    if (angle > 0)
angle -= DPI;
    else
angle += DPI;
   }
 return angle;
}
/*****************************************************************************
 FUNCTION:  normPIover2()
 PARAMETERS: double angle     ---- the angle to normalize
 PURPOSE:   normalize the input angle between -PI/2  and PI/2
 RETURNS:   double:  the normalized angle in radians
 COMMENTS:  This was designed specifically for parabola calculations
*****************************************************************************/
double
normPIover2(double angle)
{
 while ((angle > PI/2) || (angle <= -PI/2))
   {
    if (angle > PI/2)
      angle -= PI;
    else
      angle += PI;
   }
 return angle;
}

/*****************************************************************************
 FUNCTION:  signedDiff()
PARAMETERS: CONFIGURATIONS directerix
        POINT  focus
```

```
 PURPOSE:  to calculate the size distance between a point and a configuration.
RETURNS: double:  the signed difference
COMMENTS:
*****************************************************************************/
double
signedDiff(CONFIGURATION config , POINT pt)
{
return (-(pt.X - config.Posit.X) * sin(config.Theta) +
(pt.Y - config.Posit.Y)* cos(config.Theta));
}

/*****************************************************************************
 FUNCTION:  defineConfig()
 PARAMETERS: double x,y,theta,kappa       --The values that define a
                                configuration
 PURPOSE:   To allocate nad assign a configuration
 RETURNS:   CONFIGURATION:a configuration
 COMMENTS:  Was called def_configuration() in MML10
*****************************************************************************/
CONFIGURATION
defineConfig(double x,double y,double theta,double kappa)
{
CONFIGURATION newConfig;

 newConfig.Posit.X = x;
 newConfig.Posit.Y = y;
 newConfig.Theta = theta;
 newConfig.Kappa = kappa;
 return newConfig;
}


/*****************************************************************************
 FUNCTION:  defineParabola()
 PARAMETERS: double xf,yf          ---defines the focus
         double xd, yd, thetad  ---defines the directrix
 PURPOSE:   To allocate assign a parabola
 RETURNS:   PARA: a parabola type
 COMMENTS:  Was called def_parabola() in MML10
*****************************************************************************/
PARA
defineParabola(double xf,double  yf, double xd,
    double yd, double td)

{
PARA newPara;

newPara.Focus.X = xf;
newPara.Focus.Y = yf;
newPara.Directrix.Posit.X = xd;
newPara.Directrix.Posit.Y = yd;
```

```
newPara.Directrix.Theta = td;
newPara.Directrix.Kappa = 0.0;
return newPara;
}

/*************************************************************************
 FUNCTION:  reverseOrientation()
 PARAMETERS: CONFIGURATION original -- the original configuration
          orientation changed by 180 degrees
 PURPOSE:   To reverse the orientation of a given configuration
 RETURNS:   CONFIGURATION the reversed configuration
 COMMENTS:  Was called negate() in MML10
*************************************************************************/
CONFIGURATION
reverseOrientation(CONFIGURATION original)
{
 CONFIGURATION reversed;

 reversed.Posit.X = original.Posit.X;
 reversed.Posit.Y = original.Posit.Y;
 reversed.Theta = norm(original.Theta + PI);
 reversed.Kappa = original.Kappa;
 return reversed;

}

/*************************************************************************
 FUNCTION:  findSymConfig()
 PARAMETERS: double a -- distance from either point to the intersection of
                  both lines determined by the two configurations
   double alpha --The angular difference between both orientations
 PURPOSE:   This function finds the symmetric configuration of a configuration
         described by alpha and a above.
 RETURNS:   CONFIGURATION: symConfig --the symmetic configuration
 COMMENTS:  Was called def_sym() in MML10
         One drawback to this function is that it is not possible to
         represent a symmetric configuration whose alpha is equal to PI.
        find_symConfig1() is used to cover these situations
*************************************************************************/
CONFIGURATION
findSymConfig(double a, double alpha)
{
 return defineConfig(a * (1.+ cos(alpha)), a * sin(alpha), alpha, 0.0);
}

/*************************************************************************
 FUNCTION:  findSymConfig1()
 PARAMETERS: double d -- distance from the origin (base configuration) to
                  the symmetric configuration.
   double alpha --The angular difference between both orientations
 PURPOSE:   This finds the symmetric configuration of a configuration
```

```
        described by alpha and a above.
 RETURNS:   CONFIGURATION  sym_config --the symmetic configuration
 COMMENTS:   Was called def_sym1() in MML10
        This function overcomes the drawback of the original
        find_sym_config() of not being able to handle the situation when
        alpha equals PI
*************************************************************************/
CONFIGURATION
findSymConfig1(double d  double alpha)
{
 double beta = alpha 2

 return defineConfig(d * cos(beta)  d * sin(beta)  alpha  0 0)


 ************************************************************************


 FUNCTION:   inverse()
 PARAMETERS:  CONFIGURATION original --the original configuration
                        in global coordinates
 PURPOSE:    To calculate the inverse of a given configuration
 RETURNS:    CONFIGURATION: the inversed configuration
                   such that:
                   original * inversed = Identity
 COMMENTS:   None
*************************************************************************/
CONFIGURATION
inverse(CONFIGURATION original)
{
CONFIGURATION inversed;

 inversed.Posit.X = -original.Posit.X * cos(original.Theta) -original.Posit.Y *
          sin(original.Theta);

 inversed.Posit.Y =   original.Posit.X * sin(original.Theta) -original.Posit.Y *
           cos(original.Theta);

 inversed.Theta = -original.Theta;
 inversed.Kappa = -original.Kappa;

 return inversed;

}

/*************************************************************************

 FUNCTION:  compose()
 PARAMETERS: CONFIGURATION *first  -- pointer to the first configuration
                  *second -- pointer to the  second configuration
 PURPOSE:   To calculate the composition of the first and second
        configurations
```

```
 RETURNS          IFIGURATION: configuration which is the
        comp(       of the first and second configurations
 COMMENTS         ,pical example of the usage of this function is to determine
        the goal position of a configuration in global coordinates. In
        such an example, the first argument would be the original
        configuration and the second argument would be the goal
        configuration in the original configuration's local coordinate
        system. The resultant third argument would then be the goal
        configuration in global coordinates. Was called comp() in MMI.10
*****************************************************************************
CONFIGURATION
compose(CONFIGURATION * first, CONFIGURATION * second)
{
CONFIGURATION third;
 double x,y, theta;
double xx,yy,tt;

 x = second->Posit.X;
 y = second->Posit.Y;
 theta = first->Theta;


 xx = cos(theta) * x - sin(theta) * y + first->Posit.X;
 yy = sin(theta) * x + cos(theta) * y + first->Posit.Y;
tt = norm(first->Theta + second->Theta);

third.Posit.X = xx;
third.Posit.Y = yy;
third.Theta = tt;

 return third;
}

/*****************************************************************************

 FUNCTION:  circularArc()
 PARAMETERS: CONFIGURATION length  --the arc length
                  alpha  --the end orientation
                  config  --pointer to the resultant configuration
 PURPOSE:   Given the arc length and alpha, to calculate the final
        configuration
 RETURNS:   CONFIGURATION: pointer to the final configuration
 COMMENTS:  The main purpose of this function is to be used in conjunction
        with compose() to form a new next(). In this case, length would
        actually be delta-s and alpha would be delta-theta.
        Circular_arc() would determine the configuration after the incre-
        mental move in the local coordinate system of the original
        configuration. Then compose() would take the original
        configuration (in global coordinates) and the incremental
        configuration (in local coordinates) to determine the
        incremental configuration in global coordinates.
```

```
**********************************************************************************/
CONFIGURATION
circularArc(double length, double alpha)
{

return defineConfig((1- (alpha * alpha)/6) * length,
      (0.5 - (alpha * alpha)/24) * alpha * length,
      alpha, 0.0);
}
```

APPENDIX C. FUNCTION NAME COMPARISON TABLES

The following tables list the function name comparisons between MML10 and MML11. When there is no comparable function, an asterisk (*) is used as an indication.

MML11	MML10
localize()	new_config()
getLinearVelocity	commanded_velocity()
restOfPath()	rest_of_path()
needsToDecelerate()	*
motionRules()	get_velocity()
pathRule()	*
stopRule	*
rotateRule()	*
spiralRule()	*
computeLineYstar()	update_delta_d()
computeCircleYstar()	update_delta_d()
updateLineImage()	update_image()
updateCircleImage()	update_image()
InitMotion()	*
MotionSysControl()	control()
updateMovement	evauate_incremental_motion()
getDistanceTraveled()	evauate_incremental_motion()
getOrientationChange()	evauate_incremental_motion()
updateTotalDistance()	evauate_incremental_motion()
SetMovement()	evauate_pwm()

Table 1: CORE MOTION CONTROL FUNCTIONS

MML11	MML10
setGoalLinVelImm()	speed0()
setGoalRotVelImm()	r_speed0()
setGoalLinAccImm()	acc0()
setGoalRotAccImm()	r_acc0()
setPathElement()	*
getPathElement()	get_line0()
stopImm()	stop0()
setRobotConfgImm()	set_rob0()
getRobotConfig()	get_rob0
setConfigImm()	set_c()
setSizeConstant	size_const()
haltMotionImm()	halt()
resumeMotionImm()	resume()
line()	line()
bline()	bline()
stop()	stop()
setRobotConfig()	set_rob()

Table 2: MOTION CONTROL RELATED FUNCTIONS

MML11	MML10
euDist()	eu_dis()
norm()	norm(), normalize()
positiveNorm()	pnorm()
negativeNorm()	nnorm()
normPlover2()	*
signedDiff()	*
defineConfig()	def_configuration()
defineParabola()	def_parabola()
reverseOrientation()	negate()
findSymConfig	def_sym()
inverse()	inverse()
compose()	comp()
circularArc()	*

Table 3: GEOMETRIC FUNCTIONS

LIST OF REFERENCES

[Book 94]Book, S., *Improving Software Characteristics of a Real-time System Using Reengineering Techniques.*, Master's Thesis, Naval Postgraduate School, Monterey, California, March 1994.

[Kanayama 94]Kanayama, Y., "Mathematical Theory of Robotics: Introduction to 2D Spatial Reasoning", *Lecture Notes of the Advanced Robotics Course*, Department of Computer Science, Naval Postgraduate School, Winter Quarter 1994.

[Kelbe 94]Kelbe, F., Private verbal communications.

[Lee 94]Lee, T., *The Stable and Precise Motion Control for an Autonomous Mobile Robot*, Master's Thesis, Naval Postgraduate School, Monterey, California, March 1994.

[Scott 93]Scott, R. C., *Reengineering Real-Time Software Systems*, Master's Thesis, Naval Postgraduate School, Monterey, California, September 1993.

INITIAL DISTRIBUTION LIST

Defense Technical Information Center Cameron Station Alexandria, VA 22304-6145	2
Dudley Knox Library Code 052 Naval Postgraduate School Monterey, CA 93943-5002	2
Chairman, Code CS Computer Science Department Naval Postgraduate School Monterey, CA 93943	1
Dr Yutaka Kanayama, Code CS/KA Computer Science Department Naval Postgraduate School Monterey, CA 93943	2
LCDR Frank Kelbe, Code CS/Ke Computer Science Department Naval Postgraduate School Monterey, CA 93943	1
CDR, C Company, 306 MI Bn Attn: Cpt Kevin L. Huggins Fort Huachuca, AZ 85613	2
Mrs. Janet H. Lester 1924 Goldsmith Lane Unit 30 Louisville, KY 40218	1
Mrs. Emma Sandoval Apartado 6-6925 El Dorado Panama Republic of Panama	1
Mr. James Huggins 303 Pow Hatan Dr. Poquoson, VA 23662	1

Mrs. I. Forrest
942 Milford Lane
Louisville, KY 40207 1

Mr. Roy Shelton Jr.
15748 St. Mary's
Detroit, MI 48227 1